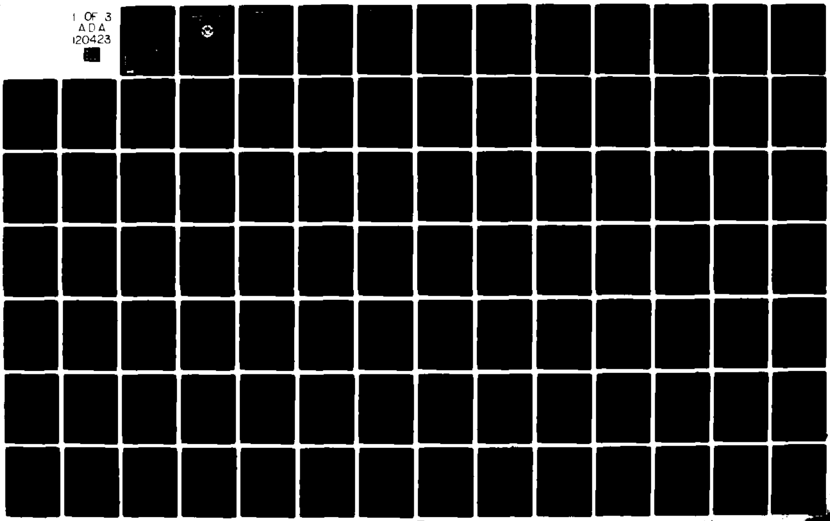
AD-A120 423 SOFTWARE ENGINEERING BASICS: A PRIMER FOR THE PROJECT MANAGER (U) STEVEN PATRICK ARTZER, ET AL NAVAL POSTGRADUATE SCHOOL, MONTEREY, CA JUN 82

UNCLASSIFIED

1 OF 3
ADA
120423



AD-A120 423

Software Engineering Basics: A Primer for the Project Manager

Steven Patrick Artzer, et al

Naval Postgraduate School
Monterey, California

June 1982

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE ENGINEERING BASICS:
A PRIMER FOR THE PROJECT MANAGER

by

Steven Patrick Artzer
and
Richard Alvin Neidrauer

June 1982

Thesis Co-Advisors: R. W. Modes
N. F. Schneidewind

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A120413	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Software Engineering Basics: A Primary for the Project Manager		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Steven Patrick Artzer Richard Alvin Neidrauer		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 279
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Engineering	Software Design
Software Quality	Software Maintenance
Software Metrics	Software Documentation
Software Management	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

→A key to any software development project is the presence of technically proficient management. The discipline of software Engineering offers many different tools and techniques to aid the project manager in the development of quality software. This thesis provides an overview of this discipline, including its goals and underlying theoretical concepts. A discussion of specific tools and techniques that are applicable

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 Continued:

throughout the life cycle is included. Recognizing that the maintainability of the software is a primary consideration of any development project, two methods of measuring software for this important attribute are examined. Among the conclusions is that there exists a need for further research necessary in order to validate the utility of the tools and techniques of Software Engineering in large scale applications.

Software Engineering Basics: A Primer for the Project Manager

by

Steven Patrick Artzer
Lieutenant, United States Navy
B.S., United States Naval Academy, 1977

and

Richard Alvin Neidrauer
Lieutenant, United States Navy
B.S. (Ed.), State College at Buffalo, 1971

Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
June, 1982

Authors: [illegible]

[illegible]

Approved by: [illegible]
Thesis Co-Advisor

[illegible]
Thesis Co-Advisor

[illegible]
Chairman, Department of Administrative Sciences

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

A key to any software development project is the presence of technically proficient management. The discipline of Software Engineering offers many different tools and techniques to aid the project manager in the development of quality software. This thesis provides an overview of this discipline, including its goals and underlying theoretical concepts. A discussion of specific tools and techniques that are applicable throughout the life cycle is included. Recognizing that the maintainability of the software is a primary consideration of any development project, two methods of measuring software for this important attribute are examined. Among the conclusions is that there exists a need for further research necessary in order to validate the utility of the tools and techniques of Software Engineering in large scale applications.

TABLE OF CONTENTS

I. INTRODUCTION

"You software guys are too much like the weavers in the story about the Emperor and his new clothes. When I go out to check on a software development the answers I get sound like 'We're fantastically busy weaving this new cloth. Just wait awhile and it'll look terrific.' But there's nothing I can see or touch, no numbers I can relate to, no way to pick up signals that things aren't really all that great. And there are too many people I know who have come out at the end wearing nothing but expensive rags or nothing at all."

An Air Force Decision Maker [1].

A. BACKGROUND

The federal government is recognized as the world's largest consumer of computer processing resources with expenditures estimated at between $10 and $20 billion dollars annually [2]. The majority of these expenditures are for the development and maintenance of software. The largest single individual consuming agency within the federal government is the Department of Defense. In Fiscal Year (FY) 1965, DoD accounted for 60% of the annual federal ADP costs. By FY 1975, DoD's percentage of the total had decreased to 50.5%, but the total federal spending had increased from $1.132 billion to $3.1 billion [3]. These figures do not reflect expenditures for software designed for use in embedded tactical computer systems. Thus, DoD's percentages of total federal computer expenditures may actually be significantly higher.

Literature on the subject of computer software development is replete with examples of projects that were delivered late, cost too much, or failed altogether. A recent GAO study [2] examined nine cases of software development projects. Of the nine cases reviewed, only one yielded software that was usable as delivered. The combined total costs and developmental times increased from estimates of $3.7 million and 10.8 years to an actual cost of $6.8 million and an actual duration of 20.5 years. For its $6.8 million, the federal government received $3.2 million worth of software that was delivered but never successfully used (47%), $1.95 million worth of software that was paid for but never delivered (29%), $1.3 million of software that was used but either required extensive rework or was later abandoned (19.1%), and only $119,000 worth of software that could be used as delivered (1.75%). While the relatively small size of the project may have been a factor, in discussing the single project that was adjudged to be successful, the GAO pointed to the presence of a highly capable, technically proficient project manager as one of the primary reasons for the project's success.

Boehm [1, 4] estimates that by 1985, 80% of the total costs of computer systems will be in software. This is in marked contrast to the situation 30 years ago where software represented only approximately 20% of the total investment. With respect to the lifecycle cost of software, evidence indicates that up to 75% of the total is in the maintenance of existing software.

A 1981 GAO study [5] on the management of software maintenance activities within the federal government estimated that approximately 67% of the total dollars expended in the federal government for non-tactical software is for maintenance. The situation in the tactical environment does not appear any more reassuring. A study issued in 1979 by the Rome Air Development Center [6] give an estimated figure of up to 75%. In another study that examined the cost of maintaining tactical software, De Roze reports that Air Force Avionics software costs about $75 per instruction to develop but over $4000 per instruction to maintain. This study serves to further highlight the rising costs of software maintenance as well as underscore the need to design software with maintainability in mind.

This "software crisis" as many authors have labeled it is not merely limited to the federal government. Mills [8] states that in the last 25 years, 75% of the total data processing personnel are used in maintenance vice development activities. Elshoff [9] indicates that a similar figure at General Motors and believes that the situation at GM is indicative of industry in general. Daly [10] has stated that 60% of General Telephone and Electric's 10-year lifecycle cost for real-time software is devoted to maintenance.

Although the production techniques for error-free complex software systems is and probably will remain beyond the state of the art in the foreseeable future, indications are that improvement in specification, design, implementation, and management techniques can reduce the amount of effort needed in the corrective maintenance of software as well as aid in the productivity of designers and programmers. In an analysis of the relative cost of correcting errors in software as a function of the phase in which they are corrected, Boehm [4] has demonstrated that it clearly pays off to invest effort in the early planning and design phases rather than waiting to discover the error during operations.

Software Engineering, a term introduced about 14 years ago [11], has come to encompass many diverse activities such as program tools and standards, design philosophies, and management techniques. The proponents of these techniques have their own, usually unique impression of what they mean by such terms as modules, structured programming, structured design and a veritable host of other "buzzwords" that have come to be associated with software development activities. Since these activities are highly individualistic and the practitioners often considering themselves as craftsmen practicing a highly developed form of black magic, it is not surprising that such a diversity of opinion regarding the terminology, conceptual underpinings and relative merits of the proposed tools and techniques exists.

B. PURPOSE AND APPROACH

A major goal of this thesis is, then, to present a concise overview of the discipline of Software Engineering. It will attempt to clarify the lexical ambiguities that have arisen by examining protions of the literature and provide definitions for many of the common terms that have been used by various authors in describing the tools and techniques that have been offered as possible solutions to the "software crisis". In doing so, it will provide potential software development project managers with a modicum of understanding of the various tools and techniques available to him in aiding the management of the project to a successful conclusions.

Chapter II provides definitions of software and software engineering. It examines proposed software engineering curriculums as a means of determining the scope of control of a software engineer. Chapter III examines and defines some characteristics of "quality" software as a means of describing the goals of the discipline. It also looks at some of the conceptual underpinings of software engineering and offers definitions of the principles presented. Although there exists some definitional controversy among various authors regarding some of the terminology, an attempt will be made to provide a consensus definition as well as point out differences where they exist.

Chapter IV examines various models of what has been called the software lifecycle and looks at some of the tools and methodologies available for use in the phases of software development. Particular emphasis is given to those techniques and practices in the requirement analysis, specification and design phases that appear to lead to more reliable and maintainable software. The preponderance of the literature suggests that attention to these issues must be given early in the development rather than waiting for the implementation phase.

Documentation is recognized as the important product of design, not merely as a by-product of coding [12]. Chapter V addresses this important issue and identifies various DoD requirements for software documentation both within the tactical environment and in the ADP community.

The evidence indicates that of the multitude of goals of software engineering, maintainability must be a primary consideration. The literature abounds with articles by authors describing various techniques to achieve maintainability. Chapter V also examines two methods to measure software and its associated documentation in order to provide a project manager with a means of determining whether or not the delivered products will be easily maintained. One of these methods was developed and is currently used by the Headquarters, Air Force Test and Evaluation Center, Kirtland Air Force Base, New Mexico evaluation teams during the operational test and evaluation phase (OT&E) of weapons system software acquisition. The other method is one proposed by Tom Gilb in his book, Software Metrics [13]. Emphasis will be focused on the evaluation of these methods as they are currently used as well as possible means of extending them for use earlier in the development cycle.

Finally, Chapter VI contains conclusions and recommendations.

II. SOFTWARE ENGINEERING

A. DEFINITION

Before attempting to define what is meant by the phrase "software engineering" it is necessary to provide a definition of the term software. Originally used in the United States around 1959 or 1960 [14], software has been used synonymously with the term computer program. A computer program is a series of instructions or statements written in an acceptable form of code that causes computer equipment to perform some operation or statement. To some authors, software is used to describe what is more commonly referred to as systems software such as operating systems, compilers, or assemblers. This usage is used to distinquish between systems programs and applications programs. To make this distinction, however, is to imply that the problems associated with the development of systems software are uniquely different from those associated with applications programs. In fact, the problems associated with the development of any large, complex piece of software are such the same regardless of its intended use.

Within the Department of Defense, there has developed a distinction between software designed to operate in an embedded weapons system and software of a more traditionally business-oriented nature. This difference is primarily the result of the enactment of Public Law 89-306, or as it is more commonly referred to, the Brooks Act. Sponsored by Representative Jack Brooks of Texas, this act places the procurement of automatic data processing equipment and components (ADPE), not including those hardware and software components embedded in weapons systems, under the administrative aegis of the Office of Management and Budget and the General Services Administration. Within DOD, ADPE falls under the cognizance of the Assistant Secretary of Defense (Controller) while weapons systems software is under the purview of the Office of the Undersecretary of Defense for Research and Engineering. A result of this branching of control is that two distinct organizational structures, each replete with its own set of instructions, directives, policies and standards have emerged. This separation has led to the view that somehow there are unique differences between tactical and non-tactical software development. While recognizing that the rules and regulations regarding the acquisition of these products are indeed different, a major thrust of this thesis is that the developmental and managerial issues are largely the same. Although the requirements for reliability and maintainability of a major defense system may be more critical than that of a payroll system, it appears clear that many of the problems associated with the designing of either are similar. The degree of criticality of either type of software will vary as to its purpose and is largely a function of how vital the software is to the user in aiding him in the performance of his mission and the importance of that mission to the achievement of the organizational goals and objectives. The tools, techniques and methodologies provided by software engineering are equally applicable to either type of software. The decision by a project manager to select a particular tool or method is one that must be resolved on a case-by-case basis by analyzing the trade-offs involved between the costs incurred by the usage of the tool versus the lifecycle benefits from their usage.

Software, as used in this thesis, is defined as a computer program or programs, including code and internal data, as well as the associated documentation required to develop, operate and maintain the programs (adapted from [4] and [15]). Although documentation is not traditionally considered as an integral component of software, it is included to emphasize the importance of the timely generation of documentation as a vital ingredient in the software development process.

Software Engineering has been defined by F. L. Baur as "The establishment and use of sound engineering principles (methods) in order to obtain economical software that is reliable and works on real machines [16]." Various authors [4, 16, 17] have made the analogy between the design and production of software and the other, more classical engineering disciplines. These disciplines have two major commonalities. First they are based on and draw their power from the use of natural laws. Secondly, they used a design methodology that allows them to detail the design process through sucessive levels of structure through the use of documentation such as blue prints or schematic drawings.

The introduction of the phrase "software engineering" as the title of a conference sponsored by the NATO Science Committee in 1968 generated a great deal of controversy. Critics have contended that unlike the other engineering fields, software engineering is not based on any natural laws. The other disciplines all deal with visible and tangible objects. Electrical engineering is the most abstract of the classical engineering disciplines since electricity is not a material. Yet even electricity, through the use of the appropriate tools, exhibits characteristics that are both visible and tangible. Furthermore, they contend that the development and design of software is more of an intuitative art than science. They claimed success for various methodologies and techniques, more noteable for the lack of standardization makes the development process more of a craft than a science.

Even some of the proponents of the phrase have recognized that to one degree or another, the analogy is not exact. Hoare [18] contends that it may be 'grossly inadequate, not to say presumptious" to compare software developers with present-day engineers. Given the recent emergence of computing vis-a-vis the other, more well-established engineering fields, software engineers and computer scientists are just beginning to formulate a set of "laws" concerning the properties of software. The fact that there has been an accumulating body of literature and attempts by both the developers and users to impose a degree of standardization to the process of software development is indicative of the appropriateness of the phrase, "Software Engineering".

B. THE SCOPE OF SOFTWARE ENGINEERING

Engineers have been characterized as problem solvers. Unlike scientists who attempt to discover new insights about the laws of the universe, engineers attempt to apply the theoretical knowledge of the scientist in the solving of real problems. Like the civil engineer who uses the theories of the sciences of physics and chemistry to design a bridge in such a manner as so it can be constructed in an economical and efficient manner, so does the software engineer attempt to design software utilizing the theoretical body of knowledge that has come to be known as computer science. This is not an attempt to denigrate the contributions of those authors who have significantly added new insight into the theory of software development and who consider themselves software engineers. In the traditional engineering fields there exists a gray area between the "pure" engineer and the "pure" research scientist. Rather it is an attempt to establish the scope of software engineering. One way of delinating that scope is through the examination of proposals for the establishment of a curriculum for the education of prospective software engineers.

Table II-1 is a model of the curriculum for software engineering proposed by Freeman and Wasserman [19, 20]. Detailed descriptions of the individual course syllabus are contained in the first reference and will not be detailed here. The authors offer seven major instructional objectives. Upon the successful completion of this curriculum, a student should be able to:

1. Do software development by accurately using at least one state of the art method in each of the four major areas of analysis, design, implementation, and testing.

2. Test and measure software by devising experiments that will validate a software system's quality; e.g. reliability, efficiency.

3. Create specialized software systems by incorporating the state of the art in at least one significant area; e.g. compilers, operating systems, data base systems, and various applications systems.

4. Manage effectively a moderate size project (3-7 people for 1-2 years).

5. Communicate effectively with users, other managers, and technical people.

6. Learn new software engineering methods rapidly and be able to keep up with relevant advances in computer science.

7. Evaluate, choose and implement new methods in a development organization.

TABLE II-1. PROPOSED SOFTWARE ENGINEERING CURRICULUM [19].

General		Management Techniques	
G1.	Introduction to Software Engineering	M1.	Management of Software Development
G2.	Creative Problem Solving	M2.	Technical Writing and Software Documentation
		M3.	Economics of Software Engineering
		M4.	Principles of Management Science

Applied Computer Science		Analytical Tool	
CS1.	Computer Systems Architecture	A1.	Software Reliability
CS2.	Language Implementation	A2.	Applied Formal Analytical Techniques
CS3.	Operating System Design	A3.	System Performance Measurement/Evaluation
CS4.	File and Database Design		
CS5.	Communication-based Systems		

Software Development	
SD1.	Software Requirements and Specifications
SD2.	Software Design Techniques
SD3.	Programming Methodology
SD4.	Software Validation and Verification
SD5.	Software Engineering Laboratory

Some of the critical assumptions of this proposal are that it is designed for a Master's level of student and that the prospective student has recently completed undergraduate work in computer science. It also assumes that the student has had some practical experience in the data processing field. Some particular advantages of this proposed curriculum is the inclusion of the managerial aspects of software development as an integral component of the skills that must be developed by software engineers. One of the most important aspects of engineering is the ability to effectively manage and communicate with personnel both within the data processing community and with the users and other managers who may not be familiar with many of the terms and techniques of software development. Implicit in Bauer's definition of software engineering is that the engineer must be able to effectively analyze the economic implications of the various design decisions facing him in order to ensure that the software developed is both economically and technically feasible and meets the needs of the sponsoring organization.

Jensen and Tonies have critized this proposed curriculum, stating in Chapter 8 of their book, Software Engineering [17], that the result of this curriculum is to produce "a better trained computer scientist but not a software engineer". They conclude that a software engineer is part generalist in the computer science field and part manager. Their counterproposal emphasizes the need to develop the managerial and communication skills of the potential software engineer as well as exposure to engineering fundamentals and physical sciences which are the fundamental elements of the more classical engineering disciplines.

The purpose of the inclusion of these proposed curriculums in this thesis is not, however, to debate the relative merits but, rather, to aid in defining the scope of software engineering. In addition to being intimately familiar with the various methods of software development, the practitioner of this discipline must have a firm knowledge of computer hardware and firmware design and operation. Further, the software engineer must possess the ability to effectively manage the development process through the entire life of the software and be able to communicate effectively with both technical and non-technical personnel in order to ensure that the software being developed meets the needs of the end user. He or she must be familiar with the various alternatives available to translate the requirements into a working system that is economically feasible and efficient.

III. GOALS AND PRINCIPLES OF SOFTWARE ENGINEERING

From the definition of Bauer's in the last chapter, it is clear that the overall goal of software engineering is to produce economical and efficient "quality" software. However, it is necessary to examine what are the characteristics of quality within the context of software development; how they interrelate; and in some cases, how achievement of one aspect of quality can only be achieved at the expense of another. The purpose of this chapter is, then, to examine the concept of quality in order to provide insight into the goals of software engineering. It will also present an overview of a number of principles developed by various authors that appear to lead to the achievement of that goal; the development of quality software.

A. CHARACTERISTICS OF QUALITY SOFTWARE

Yourdon [21] delineates seven desirable qualities of a good program. They are, in decreasing order of importance:

1. The program works and is readily observable.
2. It minimizes testing costs.
3. It minimizes maintenance costs.
4. Flexibility - Ease of changing, expanding or upgrading the program.
5. It minimizes development costs.
6. It possesses a simplicity of design.
7. It is efficient.

Wulf [22] has identified and provided concise definitions of seven important and reasonably non-overlapping attributes of software: maintainability/modifiability, robustness, clarity, performance, cost, protability and human factors.

Ross et. al. [23] have identified four goals: modifiability, efficiency, reliability, and understandability; seven principles: modularity, abstraction, localization, information hiding, uniformity, completeness, and confirmability; and five aspects of what they call the fundamental process: purpose, concept, mechanism, and usage, to define the discipline of software engineering.

Boehm et. al. [24] have developed a Software Quality Characteristics Tree in terms of the utility of the software to the user. As Figure 3-1 shows, they have divided general utility into two catagories, as-is utility, which is a measure of the usefulness of the software as it currently exists and maintainability which, within the context of their study, is a measure of the degree to which the software can be changed as the user's objectives and requirements evolve. They identify seven high level characteristics and refine the definition of each in terms of twelve primitives for which they, in turn provide definitions and candidate metrics to measure the software to determine to what degree it possesses each of the characteristics. The reason that Figure 3-1 has been presented is to emphasize the interrelationships between various characteristics of

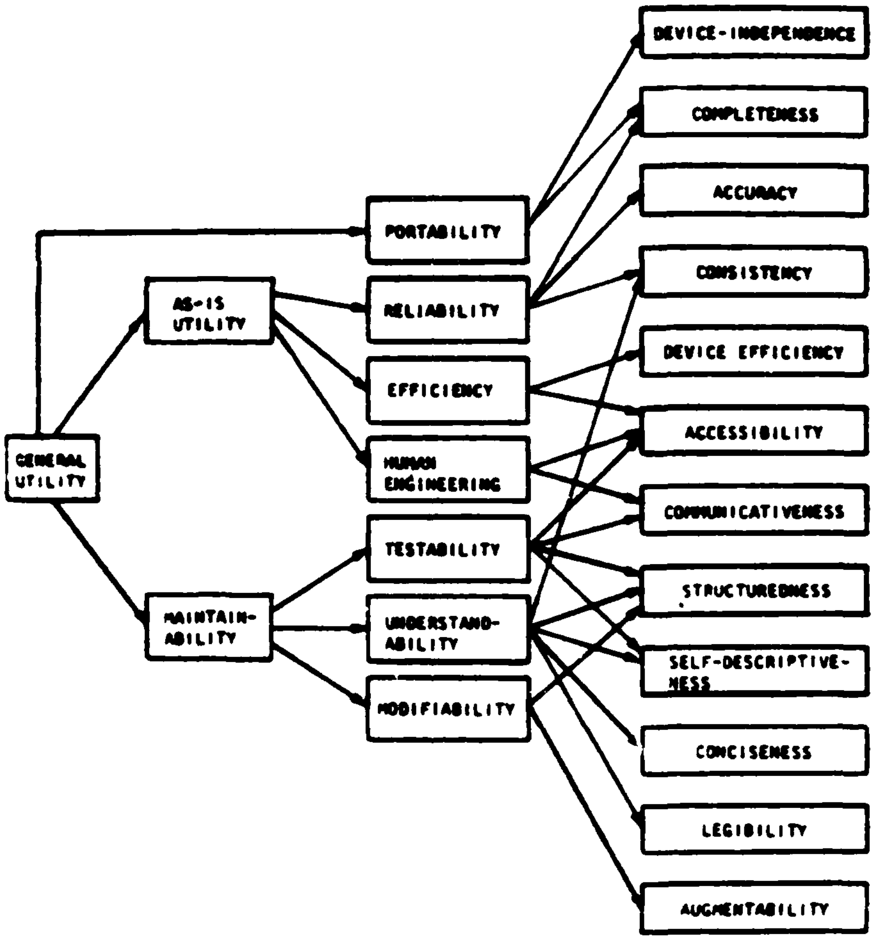


Figure 3-1. Software Quality Tree [24].

quality software. For example, maintainability requires that the user be able to understand, modify and test the program and is aided by good human engineering but is not dependent upon the efficiency or portability, except to the extent that the user's system is undergoing evolution. Furthermore, the modifiability of a program is related to the degree of structuredness and augmentability it possesses.

From the above discussion, it is clear that there exists no single universally agreed upon set of quality attributes among the advocates of software engineering. For the purpose of clarification and discussion, the goals will be presented under four major headings: Reliability, Simplicity, Evolvability, and Efficiency.

1. Reliability

Reliability is a specific measure of software quality. It is the probability of the software operating without error for a specified period of time and under specified operating conditions. The operating conditions include the hardware suite, operating system, inputs and environment (e.g. tactical operations). It is important to note that this definition distinguishes the concept of reliability from correctness, which is the degree to which the requirements of the software, as outlined in the formal specifications, are met.

Many authors object to the use of the term reliability in conjunction with software. This is primarily a result of the practice of using hardware related measurements, such as mean time between failures or mean time to repair, to indicate the reliability of software. Littlewood [25] has pointed out, for example, that those researchers who attempt to use such measures fail to account for the differences between hardware and software. As an example, he states that if a program is bug-free, its mean time to failure approaches infinity. Yet that same program may fail to meet its specifications, and is, therefore, unreliable. His contention is that because of the significant differences between hardware and software, the adaptation of hardware measures is unjustified.

Schneidewind and Kline [26] also provide a detailed comparisions of the differences between hardware and software reliability concepts as delinated in Table III-1.

Two other software quality attributes are often discussed in the literature in conjunction with reliability: correctness and robustness. While all three are discussed as though they are partially synonomous, there is one significant difference between them. Reliability is the "concept of successful operation over a specific period of time and the ascription of a probability to that success [17]." There are several models, based on error depreciation profiles, that allow reliability to be measured. It is a concept that is probabilitistically based. Correctness and robustness are, on the other hand, more subjective in nature. Hence, they are less amenable to quantification.

TABLE III-1. HARDWARE/SOFTWARE RELIABILITY COMPARISON [27].

	Hardware Reliability	Software Reliability
Fundamental Concept	Hardware fails due to physical effects	Software fails due to program error
Life Cycle Causes		
Design	Incorrect physical design	Incorrect program design
Production	Quality Control problems	Incorrect program coding
Use	Degradation and Failure	Undetected program errors
Use Effects	Hardware fails or wears out	Software does not fail or wear out
Design is function of	Physics of failure	Programmer skill
Domains	Time	Time and data
Time Relationship	Bathtub curve	Decreasing function
Mathematical Models	Theory well established	Theory not well established
Time Domain Functions	$R = f(\lambda,t)$ Exponential (constant λ) Weibull (increasing λ) Normal (wearout)	$R = f$ (errors, t) No agreed upon time function models. Various models proposed
Data Domain	No meaning	Errors - f(data, tests)
Growth Models	Several models exist	Several models exist
Metric	λ, MTBF, MTTF	Error rate, no. of errors detected or remaining
Growth application	Design, prediction, TAAF	Design, Prediction, TAAF
Prediction techniques	Block Diagrams, Fault Trees	Path Analysis, complexity, simulation
Test and Evaluation	Design and Production acceptance	Design acceptance
Design	MIL-STD-781C (exponential) other methods (non-exponential)	Path testing, simulation, error seeding, Bayesian
Production	MIL-STD-781C	None
Use of Redundancy		
Parallel	Can improve reliability	Does not improve reliability
Standby	Automatic fault detection and switching	Automatic error detection and correction
Majority logic	m out of n	Impractical

a. Correctness

Correctness means that the software is correct if it performs properly the functions it was intended to perform and has no unwanted side effects [28]. Implicit in correctness is a term known as a proof-of-correctness. Dijkstra has stated that testing can be used only to show the presence of bugs but not their absence [29]. Since testing is inadequate to ensure correctness, then, it is necessary to "prove" the correctness of a piece of software. In using proofs of correctness, the designers and programmers make use of formal specifications of the program's intent that are written in a formal assertion language and mathematically proven using a formal logic method such as predicate calculus. An input assertion defines the input domain of the program. An output assertion defines the domain of possible outputs in terms of the computation the program is intended to perform. Starting with the input assertions, the individual responsible for the proof utilizes various mathematical theorems to prove that the output assertions are satisfied whenever the input data meets the conditions imposed by the input assertions.

Various authors, such as Parnas [30] and Liskov [31] have descrived possible notation for formal specification lanaugages. Several benefits can be derived by the use of these formal languages in proving the correctness of a program. Foremost is that by using this technique a designer can verify the correctness of intermediate design decisions when they are made rather than wait until the design is complete, thereby reducing the cost of making the corrections. The proof of correctness process also forces the designer to analyze sections of his software that otherwise might only receive cursory attention. It aids in clarifying the dependencies of the program on other programs by creating an awareness of what assumptions about the input data are implicitly made in the other sections of the system. Finally, by formally specifying the input and output parameters in a mathematical notation, the ambiquity inherent in the use of natural languages, such as English, is removed.

However, as a matter of practicality for the project manager, there are also several disadvantages to using formal proofs of correctness. The process is a labor-intensive, time consuming process. Even for small relatively simple programs, the symbolic manipulations can be extremely complex. This complexity, in turn, can lead to errors in the proof, thus making the process self-defeating. Glass [32] points out that there has been little success in using this technique on programs of any significant size. Furthermore, it often requires several times the amount of work to prove a program correct than was required to write it, adding 100 to 500 percent to the cost of the software being proven. It also requires a level of mathematical sophistication than even the proponents of its usage recognize is not possessed by many of the designers and programmers currently practicing in the field. The ultimate goal of the advocates of this technique is to develop a formal assertion language that can be used in a computer program that will automate the proving process. Although some of the advantages of this technique, such as removing the ambiquity from specifications and forcing the designers to rigorously analyze and explicitly express the assumptions regarding the input and output parameters of the program can aid in the development of more reliable and maintainable software, there is little practicality in attempting to prove the correctness of a program until the process can be automated.

b. Robustness

Robustness is the degree to which a program can continue to perform its intended function in the face of hardware failures, bad inputs by the user, unexpected demands, and the incorrect operation of parts of itself [28]. It is important to stress that correctness alone is not sufficient for high quality software. A perfectly correct program that does not check inputs to ensure that they are within range may proceed to overwrite a valuable file, producing a highly unreliable behavior. As Freeman [28] states "certainly a program must be largely correct before we call it reliable but this is only a necessary condition not a sufficient one." Robustness is clearly an essential ingredient of high quality software.

One approach to increase the robustness of a program is to include in the design of the software what Parnas and Wurges [33] have called "unexpected event handlers." They use the phrase unexpected event as opposed to error because the term error implies that it should be corrected as opposed to handled. They contend that regardless of how well the original software was designed, unexpected events c n still result from device failure, incorrect or inconsistent data supplied by the user, errors resulting from changes in the program that occur from inadequate testing or even designer error. They advocate the use of software "traps" to inspect and compare data as it arrives at a program or subprogram to ensure that it meets the specifications that were formally developed. If an undesired event occurs, control of the program is turned over to a separate program called an unexpected event handler. The reason for having the unexpected event handler as a separate program rather than embedding it within the application program is to allow for easier changing of the error handling mechanism should the user later decide to change the way he desires the system to handle a particular class of errors. There should exist at least one such handler for each class of unexpected event. They provide several catagories of possible classes. The handler then takes some predetermined action such as return control to a higher level module for resolution, provide an error message, in a format designated by the user to ensure understandability, or possibly even ignores the error depending on the severity and the design of the handler. The details of this approach to ensuring robustness are contained in [12, and 33]. The point of including this approach is that Paranas and others, such as Ross [23] agree that robustness, in particular, and reliability, in general, must be considered early in the design of software, not added on by a programmer during implementation as an afterthought.

2. Simplicity

The essense of simplicity is that the nature of the software should be such that the nature of the user's interaction, whether he is the end user or maintenance programmer, should be simple and easy to use. Various terminology has been offered to describe simplicity.

Wasserman [34] defines user-friendliness as the characteristic of a system such that the system should provide meaningful error messages, should not terminate abruptly, and should enable the user to work with familiar terminology and concepts. Another phrase that has a similar definition is human factors, as defined by Boehm [24].

Invisibility is the degree to which the software masks the underlying characteristics of the computer, its operating system and the programming language used for writing the software from the user.

Another aspect of simplicity is understandability. The user is able to understand the system to the degree to which the software is well-documented, the variables contained therein are self-descriptive, and the program itself is free of complex control structures and uncontrolled branching.

Perhaps the most intuitive way to comprehend the concept of simplicity is by examining its opposite, complexity. Various measures of complexity have been offered. These measures of complexity are then used to predict the number of errors that can be expected in order to aid the individuals responsible for the testing of the software to determine where to concentrate the testing effort.

A model has been developed at the Naval Postgraduate School, supported by the Naval Air Development Center [35, 36]. Working under the hupothesis that the ease of debugging and testing is related to structural complexity, the researchers developed simulation and analytical models to measure the relationship between error detection capabilities and program structure.

In the model, a program flow is represented as a directed graph consisting of various nodes and arcs. The nodes represent merge and/or branching points within the program and the arcs represent sets of sequentially executed instructions between the nodes. Each test input defines a unique path from the start node to the exit node. The program is executed and the input proceeds along its predetermined path until an error is detected. The error is then corrected and the program is restarted along the same path. Various statistics are collected on the number of errors detected over a fixed time, the percentage of arcs traversed by one or more input, the number of errors detected and the percent of errors remaining. Complexity is measured in terms of the number of nodes, paths, arcs and source statements as well as in terms of the correctability and accessibility. The authors suggest several uses for the model. Comparing the error detection characteristics of several design alternatives can enable the project manager or his representative to select the design that will be the least costly to test. Also the model may be used to indicate where additional testing is required by increased program complexity. In order to use this model, it is necessary to perform a preliminary design of module structure.

In a related work, McCabe [37] identified the "cyclomatic" number (i.e., the number of independent circuits in a directed graph) as a software complexity metric. From this measure he derives the number of independent paths in a program. His claim is that program complexity can be reduced by limiting the cyclomatic number to a maximum of ten per module.

Underlying both of these approaches is the concept of structural complexity. Bohm and Jacobini [38] have mathematically proven that three basic control structures are sufficient for expressing any program logic: sequence, selection (IF THEN ELSE) and iteration (DO WHILE). This development is the basis for much of the philosophy developed by Dijkstra and others to control the complexity of programs. These control structures are often expended to include "DO UNTIL" and "CASE" type constructs and are illustrated in Figures 3-2 and 3-3. DoD has limited software developers to the use of these control structures in software developed for tactical weapons systems in MIL-STD-1679. A more comprehensive examination of Dijkstra's concepts of program and design structuring will be presented in the next chapter. The elimination of uncontrolled brachning by reduction of the use of "GO TO" statements greatly aid in the reduction of complexity, thereby reducing the probability of errors and adding to the understandability of software.

3. Efficiency

As Ross [23] points out, efficiency is a much abused goal, usually because in an excess of zeal, it is prematurely permitted a high priority in software engineering trade-offs. One of the reasons that it is often abused is that efficiency is normally considered within the context of the efficient utilization of hardware resources.

Boehm [24] defines efficiency as the extent that a software product fulfills its purpose without a waste of resources. Within the context of his definition, resources are considered in a narrow sense, hardware related; i.e.,

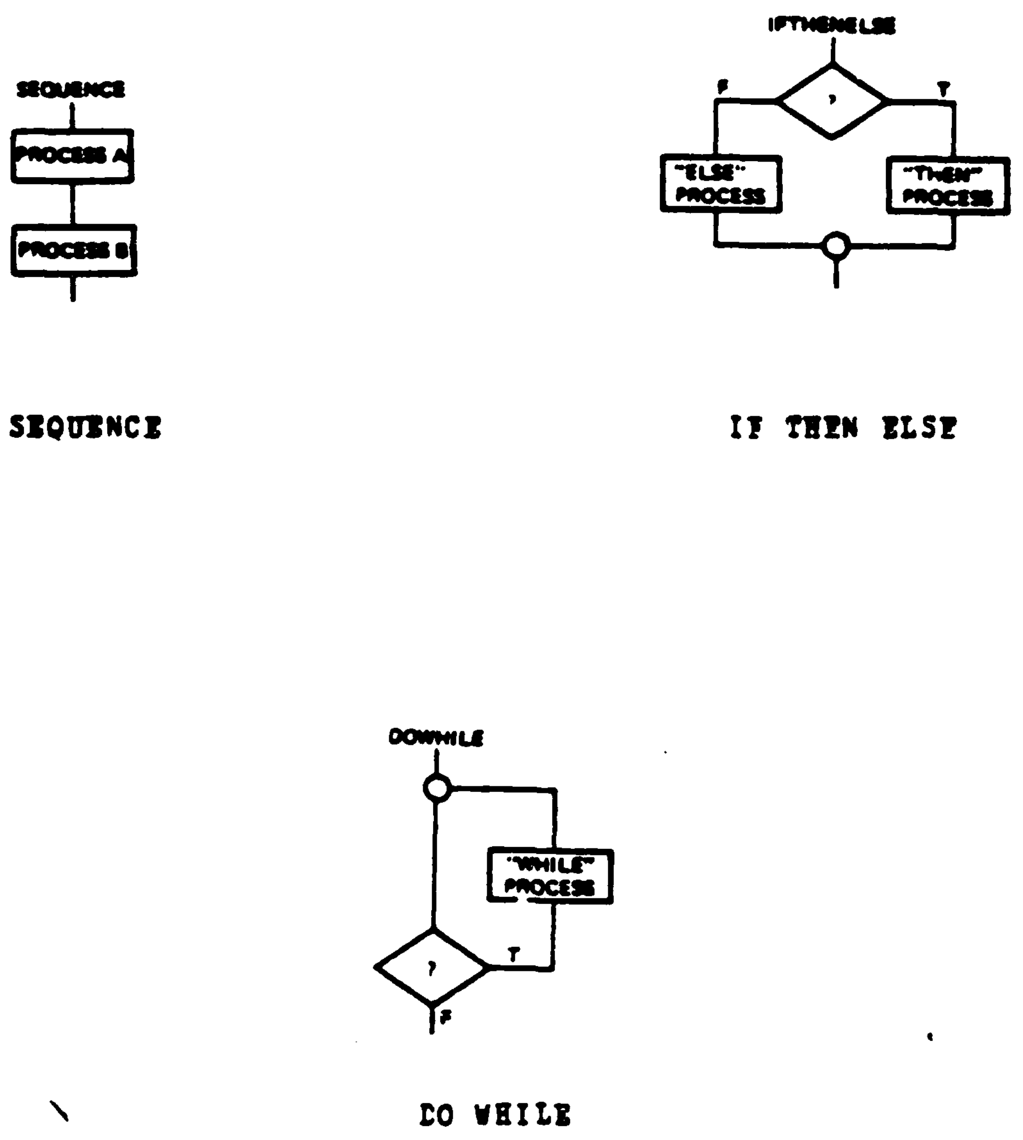
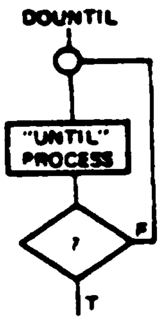
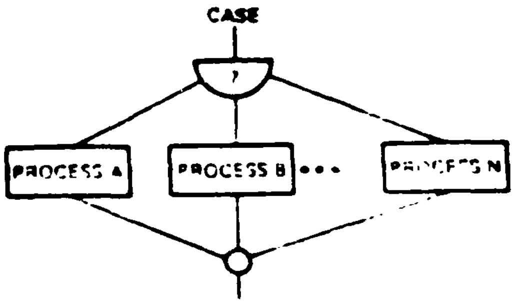


Figure 3-2. Basic Control Structures.



DO UNTIL



CASE

Figure 3-3. Additional Control Structures.

memory, the total number of instructions executed per iteration or per case, or mass storage utilization. He points out that efficiency is often achieved in opposition to some of the other goals and characteristics described previously. He contends, for example, that efficiency is "generally at the opposite end of the spectrum from portability." [24] Efficiency is often highly machine and language dependent.

Measurement of efficiency is usually done in terms of memory utilization and executive speed. Memory utilization involves performing the least amount of memory space to house both the instructions and data base [10]. Executive speed involves performing the required functions using minimum time during execution. [10]

Much of the concern for efficiency stems from the period when hardware cost factors in computer system development predominated. Since hardware represented the major portion of the total cost, the concern of designers was naturally focused on means of developing software that utilized the available hardware to minimize the total cost of operating the system. However, as stated previously, the relative costs of hardware and software have been reversed due primarily to advances in hardware development and production technology. Maintenance costs have been identified as the largest single cost factor in the entire development process. Given this reversal, efforts to minimize the usage of hardware resources that are done at the expense of maintainability appear to be misguided.

The use of low level languages such as assembly language is the most common technique used to ensure efficient use of hardware resources. Assembly language is a programming language in which there is a one-to-one correspondence between each assembly language statement and a machine language statement. Machine language is the binary representation of instructions that a computer actually executes. Assembly language statements use a more readable alphanumeric symbology which suggest the statement's function. Since there is a direct correspondence between assembly language and machine language instructions, the programmer can control the use of memory registers and other parts of the computer hardware directly, thereby having control over the amount of each component the program uses. High level languages, in comparison, use a compiler or translator to translate the high level instructions into machine language instructions. Each high level instruction requires several machine level instructions to perform its task. Thus the compiler vice the programmer has control over what resources are used to execute the program. In addition, the machine language instruction sets vary widely, both in format and numbers, and thus, assembly languages vary widely among different manufacturers. This means that program written in assembly language for a particular manufacturer's equipment must be completely reprogrammed before it can be run on a new manufacturer's machine. Thus, the use of assembly language directly restricts the portability of the software.

At the NATO Science Committee conference on Software Engineering techniques, which was the successor to the original 1968 NATO conference, Lang [51] listed two other "grave disadvantages of assembly language:

- Apart from the few who delight in such intricacies, most people find assembly language programs harder to write, read, understand, debug, and maintain than high level lanaguages.

- It provides the poorest conceptual framework for the programmer to express the computing operations he wants performed." [51]

Yourdon [21] asserts that in addition to the fact that assembly languages often prevent the usage of good structured programming techniques by the allowing of unconditional "jump" statements, whcih are the equivalent to GOTO statements in high level languages, "...high level languages, e.g., ALGOL, PL/I and COBOL have the potential for being more efficient with the structured programming approach." All of the previous discussion is not intended to negate the need for the efficient use of the resources. Blatant inefficiency cannot, of course, be tolerated. In some cases, particularily in the development of software for embedded computer systems, efficiency issues can be critical. Response time, which is a function of the execution speed of the program amoung other things, for a missile defense system may well be the single most critical factor in determining the success or failure of the entire system. Similarly, computers embedded on aircraft platforms are often very limited as to the amount of memory that is available.

However, efficiency may be more properly treated within the context of a broader definition of resources. Efficiency should also address the efficient use of the individuals who design, implement, maintain, and utilize the software. Ross [23], for example, contends that efficiency questions are "best treated within the context of other issues. For example, achieving a high degree of modifiability can provide the basis for meeting efficiency goals during the tuning phase of software development."

Brooks [52] points out that programming productivity, a measure of the efficiency of the personnel who are responsible for the implementation of software designs, may be increased as much as five times when a suitable high level language is used.

Glass [32] describes the benefits from programming in high order languages in terms of both productivity and reliability. He points out that high level language code requires fewer statements than assembly language. Therefore, there are fewer chances of error. Furthermore, the HOL programmer is screened from an entire class of possible errors related to hardware intricacy since the compiler accomplishes the task of making hardware dependent choices.

Thus, the programmer can concentrate on the application problem he is solving rather than the hardware on which the solution will run.

With respect to maintainability, often the programmer responsible for program maintenance is not the individual who originally designed and implemented it. There is a significant difference in the readability of programs written in higher level languages as opposed to assembly languages. The high level language statements are more English-like and therefore more understandable. Assembly language statments are generally more abbreviated and require greater effort to comprehend. Also, the increased complexity and the resulting potential for errors resulting from the greater degree of detail associated with the hardware-software interfaces may significantly increase the effort devoted to corrective maintenance. Lientz and Swanson [41], in reporting the results of a survey of commercial organizations that have data processing facilities, found that approximately 70% of all maintenance activity at these organizations is in the area of enhancement vice corrective maintenance. Clearly the use of high level languages can increase the maintainability of software as well as allow maintenance programmers to concentrate on making adaptive and perfective changes and not devote much of time and effort in correcting implementation errors.

Perhaps the most eloquent argument against the emphasis on efficiency as it is most commonly thought of is that offered by W. A. Wulf and cited in Reference [21]:

> "More computing sins are committed in the name of efficiency (without measurably achieving it) than for any other reason. One of these sins is the construction of a 'rat's nest' of control flow which exploits a few common constructive sequences. This is precisely the form of programming that must be eliminated if we are ever to build correct, understandable and modifiable systems." [53]

In conclusion, it is not the purpose of this thesis to advocate inefficient practices in the development of software systems. Rather, it is the contention that efficiency must be looked upon in a more global context. Each project or case must be decided individually based on the constraints involved. The question of efficiency must, however, be examined within the context of the efficient utilization of all of the resources required to develop, operate and maintain the software, not merely just one aspect.

4. Evolvability

Recognition that the majority of the effort and cost of software development is devoted to the changing of software after its initial acceptance and operation has led to the inclusion of evolvability as one of the key goals of software engineering. Evolution implies change over time. Just as an organization's missions and requirements change, the software designed to aid the organization in fulfilling its requirements must also be amenable to change. Many of the tools and techniques that have been proposed have emphasized the need to design software in such a manner as to facilitate changes to it. The literature that has been presented to advocate these methodologies has offered various phrases to describe the characteristic of software quality that allows it to be changed easily. They can be summarized under three major headings: maintainability, portability and reusability.

a. Maintainability

There exists no universally accepted definition of maintainability. Some authors, such as Lientz and Swanson [39, 40] take a broad, inclusive view of maintainability. In an attempt to provide a basis for the understanding of the "dimensionality" of the maintenance problem, Swanson [40] differentiates between three types of maintainability: corrective maintenance, adaptive maintenance and perfective maintenance. Corrective maintenance is performed in response to failures such as failures to meet performance criteria or abnormal stopping of a program. Adaptive maintenance is performed in response to changes in the environment. Perfective maintenance is the activity performed to make the software a better implementation of the design, such as improving processing efficiency or to add new features.

Other authors such as Myers [41] and Tausworthe [42] take a more restrictive view towards what constitutes maintenance activities. Myers [41] defines maintainability as "a measure of the cost and time required to fix software errors in an operational system." He differentiates between maintainability and adaptability, which is defined as "a measure for the ease of extending the product, such as adding new user functions to the product." [41]

Tausworthe [42] concurs with Myers' view of maintainability and contrasts it with both adaptability and modifiability. He defines these terms as:

"Maintenance: Alterations to software during the post-delivery period in the form of sustaining engineering or modification not requiring a reinitiation of the software development cycle.

Sustaining Engineering: Software related activities in the post-delivery period, principally supportive in form, which keeps the software within its functional specifications.

Adaptation: Modification of existing software in order that it may be used in a program development, as opposed to developing another module for that same purpose.

Modification: The process of alterning a program and its specifications so as to perform either a new task or a different or similar task." [42]

Thus, while Swanson takes a more expansive view of what constitutes software maintenance, Tausworthe, with his narrower scope, would exclude the two latter catagories as outside the scope of what constitutes maintenance activity, placing them into the catagory of modification. The General Accounting Office [5] has cited the lack of a uniform definition as to what constitutes maintenance activities as one of the primary reasons for the absence of software maintenance management within the federal government.

Underlying this lack of uniformity is that, in Kline's view [27], much of the terminology used in describing software maintenance has been inappropriately adapted from hardware maintainability concepts. As was the case with reliability, many authors have tried to apply concepts of hardware maintainability to the maintenance of software. In hardware maintenance, there exists two broad catagories; preventive and corrective maintenance. Preventive maintenance is that maintenance performed, on a scheduled or on-condition basis, for the purpose of retaining an item in a satisfactory operating condition [26]. Implicit in this concept is the notion that hardware physically degrades over time. By routine inspection and servicing, its failure can be prevented and the life of the equipment prolonged. Software, on the other hand, does not physically wear out with usage. Thus the concept of replacing software with a "spare" module is inapplicable. Authors, such as Glass [32], who use the term preventive maintenance in asscciation with software are actually describing various design techniques such as modularization, program structuring, parameterization and documentation that make corrective maintenance of software easier. In order to minimize the confusion with hardware maintainability, Kline suggests substituting the phrase "software configuration management" to emphasize the difference between the two concepts as well as to emphasize the importance of configuration management as a tool for controlling change to existing software. [27]

Rather than present further definitions of maintainability, it will simply be stated that software maintainability as used in this thesis will refer to the characteristic that a software product possesses to the extent that it facilitates updating to satisfy new requirements or can be modified to correct mistakes (adapted from [24]). The point to be made is that maintainability, like reliability, is an issue that must be addressed early in the development cycle and not postponed until after the design has been completed. Since maintenance is both a process for the correction of errors and also of adapting to new organizational requirements, the tools and techniques adopted during the development of software must facilitate the economical and efficient adaptation of the software to meet the new requirements as the organization's needs evolve.

b. Portability and Reusability

Poole and Waite [43] define portability as the "measure of the ease with which a program can be transferred from one environment to another." Closely allied with this concept is the concept of reusability. Reuseable software is "existing software, including specification, design, code and/or documentation, which can be employed, in part or total, into a new end use." [44] This definition of reusability emphasizes one specific approach, which is to reuse the specification and design as well as the code itself and highlights the importance of documentation as a means of identifying software that has potential for reuse. It also emphasizes the need to techniques of specifying and designing software that can be readily modified for reuse in the new applications.

The most common approach to portability is the use of high level languages, such as COBOL or FORTRAN for which compilers exist in most computers. The software that has been used in the present equipment configuration is recompiled using the compiler of the new ocmputer to create machine interpretable object code that can be executed on the new machine. An outgrowth of this approach is the limitation on the programming languages that may be utilized by developers of software for the Department of Defense. DoDINST. 5000.31, "Interim List of DoD Approved Higher Order Programming Languages (HOL)," limits, with certain exceptions, developers to the use of six approved languages, CMS-2, SPL-1, TACPCL, JOVIAL, COBOL and FORTRAN. SECNAVINST. 5236.1A, "Specification, Selection, and Acquisition of Automated Data Processing Equipment," limits developers of non-tactical software to the use of Federal Standard COBOL for business and logistics applications and American National Standards Institute (ANSI) FORTRAN for scientific or engineering applications. By attempting to reduce the proliferation and use of other HOL languages, these instructions are aiding in efforts to increase the portability of DoD software.

In an effort to further limit and standardize the number of approved HOL programming languages, DoD has adopted a common programming language based on the language PASCAL to use as its future HOL for embedded computer software [45, 46]. It is named after Ada Augusta who is generally credited as having been the first programmer as an assistant to Charles Babbage, and is called, appropriately enough, ADA. The development of one common programming language for tactical software clearly has the potential for improving the portability, reusability and maintainability through standardization of the language. Furthermore, by utilizing a single standard language, maintenance programmer productivity could conceivably increase since they would only have to be familiar with one language instead of several.

ADA is not, despite these apparent advantages, universally accepted in its present form. Dijkstra [47] has criticized ADA as being neither "complete nor concise" and has expressed concern over its size by pointing out that ADA's list of reserved words amounts to "more than ten percent of Basic English." Also Reference [46] indicates that the Department of the Navy has not been an enthusiastic support of this proposed standard, largely due to the large number of programs and programmers that have utilized CMS-2, which is the Navy's primary tactical software language. Despite the lack of universal support, the ADA project is continuing. The Army is the lead service in this project and is currently attempting to develop a prototype compiler for ADA. The original due date for testing of this compiler was April, 1981. However, due to problems in development, this target has not been met.

As is implied by the definition of reusability, another approach besides the use of standardized languages is to consider requirements for portability and reusability during the development of the software. Among the major conclusions contained in Reference [44] is that

> "Reusability must be addressed as an objective of the development process. One can not build software and then decide (after the fact) they intend to reuse it. At the time the development is started, specific guidelines for methods and standards must be established to support the intended reuse."

In the area of design this group has recommended that standards for interface specification and functional design techniques that focus on the identification, structure and partitioning for reuse are required to achieve a greater degree of reusability. [44]

One approach to a design philosophy to develop reusable software has been put forth by Parnas in References [48, 49]. He contends that a software designer must be aware that he is not designing a single program but rather a "family" of programs. A set of programs is considered a program family as they have "so much in common that it pays to study their common aspects before looking at the aspects that differentiate them." [49] Some of the ways that the members of a program family may differ are:

1. They may run on different equipment configurations.

2. They may perform the same function but differ in the format of input and output data.

3. They may differ in certain data structures or algorithms because of differences in the size of the input sets.

4. They may differ in certain data structures or algorithms because of differences in the available resources.

5. Some users may require only a subset of the services or features required by other users.

He offers a four part methodology to build software systems that are more readily reused. The first part is to identify the "subsets" of a system that are candidates for later reuse during the requirement definition. Criteria such as differing equipment configurations within an organization or functions of a general utility, such as sorting programs, are two possible evaluation considerations that might be used in identifying candidates for reuse. In identifying various subsets, the designer must first identify the minimal subset of a system that might conceivably perform a useful service. While recognizing that this minimal subset is not likely to be worthwhile to develop by itself, it should be useful to include this subset as a part of a larger system. The designer then identifies minimal increments to that subset. The emphasis on minimality is to ensure that each component performs a single function.

The second part of his methodology involves the precise definition of the modules and the interfaces between them. The modules are designed to localize the effects of the parts of the system that are likely to change. The interfaces should be designed to be "insensitive to the anticipated changes." [49]

The third part of the Parnas methodology is utilization of the virtual machine concept, also referred to as an abstract machine [43, 50]. This concept is to design interfaces and modules that do not distinguish between those functions that are implemented in software and those implemented by the hardware or operating system. The goal of this concept is to produce a design that is truly machine independent. The virtual machine serves as a model of several potential real hardware/software implementations. Thus, it must be emphasized that at the time of implementation, there must be a decision as to whether the functions will be provided by the hardware chosen or that it would be necessary to provide for them via software. By postponing this decision until implementation, however, the development process is able to proceed without having the final hardware configuration identified.

Another advantage of designing abstract machine systems is that it allows the program to be implemented on a wide variety of equipment configurations with the implementation process determining which set of instructions or functions are available from the equipment/operating system and those which must be included in the software.

The fourth and most crucial step is in identifying what Parnas calls the "uses" hierarchy, or structure. A program "A" is said to use program "B" if A requires the correct execution of B in order to complete its task as described in the specifications. "Uses" can also be formulated as "requires the presense of." [49] A key feature of the uses hierarchy is that its structure is restricted to prevent looping from a lower level in the hierarchy to a higher one. Level 0 of the hierarchy is the set of all programs that uses no other program. Level i (where i is greater than or equal to 0) is the set of all programs that uses at least one program on level (i-1). By identifying this hierarchy, which is an execution as opposed to a design hierarchy, Parnas contends that the identification of useful subsets is readily apparent.

The concept of reusability has sparked a great deal of interest within the federal government. In addition to the conference referenced earlier, there has been established a Federal Software Exchange Center (FSEC) under the aegis of the General Services Administration to promote the sharing of common-use software and related documentation among users of non-tactical software. The procedures and requirements are detailed in the Federal Procurement Management Regulation (FPMR) subpart 101-36.16. Agencies acquiring software are required to deliver copies of the source code listings and other documentation to the FSEC, which develops abstracts that are made available to potential users who are, in turn, required to certify that there is no available reusable software that can meet its requirements prior to contracting for new software development. The use of this repository has been limited due to the poor documentation practices of some contractors that make it difficult for potential users to determine exactly what functions the software is designed to perform and under what operating environments. Additionally, neither the FSEC nor the original acquiring agency is required to certify the quality of the product. Thus, potential reusers are leery of stating that they do not need to develop new software when they can not be sure of the quality of the reusable variety. Another factor limiting the use of this organization is the absence of any requirement to update any versions of the source code or documentation as changes are made, either to correct original deficiencies or modification of existing functions.

Clearly the issues of portability and reusability have implications for the project manager. In addition to being a consideration in the development of new software, the reuse of existing software of known quality can reduce the risk of new development as well as reduce the time and cost of the development process providing a means for the certification of the quality can be found. Although Reference [44], the Joint Logistics Commanders report on reusability, contains numerous conclusions and recommendations that are too lengthy to repeat in their entirety in this thesis, several are particularly worthy of mention.

1. The need for development of standards for requirement analysis and formal specification languages should be pursued.

2. Functional design techniques must be defined to focus on identification, structure and partition for reuse. The design must be catalogued and be accessible independent of the code.

3. Research and development of new support tools specifically to promote reusability are needed. Such new areas include specification generation and dissemination, verification and certification of reuse components, library methods and tools for cataloging and access of software and management tools for project control.

4. Incentives should be provided for DoD project managers and contractors for compliance with reusability concepts.

B. UNDERLYING PRINCIPLES OF SOFTWARE ENGINEERING

Having examined the goals of software engineering within the context of the characteristics of quality software, it is now appropriate to examine some of the underlying principles that have been found to aid in the achievement of that quality. As is the case of the goals, there is a high degree of interrelationship among the principles that assist in achieving quality software: Modularity, Abstraction, Hierarchical Design Approaches, Uniformity, Completeness and Confirmability.

1. Modularity

The division of a program or system into smaller units, or modules, is one of the oldest concepts in computing. Yet it is also one of the current trends in software engineering.

Modularity is the idea of reducing large, complex systems into smaller, more intellectually manageable parts. Various definitions of modularity have been presented in the literature. Myers [54], for example, offers the following definition:

> "Modularity is not simply the arbitrary division of a large program into smaller parts or modules. The primary goal should be to decompose the program in such a way that the modules are highly independent of each other."

D. T. Ross and others [23] define modularity as dealing with "...how the structure of an object can make the attainment of some purpose easier. Modularity is purposeful structuring."

Liskov [31] describes modularity in terms of the structural properties possessed by modular systems in the following definition:

> "Modularity consists of dividing a program into subprograms (modules) which can be compiled separately but will have connections with each other. A definition of 'good' modularity must emphasize the requirement that modules be as disjoint as possible."

While all of the above definitions have much in common, there exists a diversity among software engineers as to the criteria that should be utilized to decompose complex systems into simpler, more understandable units.

The classic approach, as typified by Myers, Yourdon and other authors associated with the concept of Structured or Composite design, is that each module should represent a single, well-defined function. Myers [55] contends that the first step in defining a module should be to describe its external characteristics. This description consists of the module name, purpose or function, parameter list, inputs, outputs and external effects on other modules within the system.

Yourdon has described two measures of modularity (originally proposed by Myers in Reference [55]), cohesion and coupling. Cohesion is "the degree of functional relatedness of processing elements within a single module" [57]. Coupling is a "measure of the strength of the interconnection between one module and another" [57]. He also describes various levels of each measure. The most desirable level of cohesion is "functional" cohesion which he defines as:

> "The strongest form of relationship between processing elements in a module; occurs when every element of processing is an integral part of, and essential to, the performance of a single function." [57]

The goal of their techniques of modularization is to strive for systems whose modules display a high level of cohesion and a low degree of coupling, or interdependence. Myers even provides a means for scoring a design that measures for coupling and cohesion in Reference [56].

An alternative set of criteria for modular decomposition are those offered by D. L. Parnas [58, 59]. Based upon the notion of reducing dependencies between modules created by shared assumptions, he has proposed a criteria that attempts to have each module encapsulate one changeable item within the system and "hiding" how each module deals with that item from other modules. This concept is called "information hiding" and the modules are known as information hiding modules.

The first step in his decomposition methodology is to identify those design decisions that are likely to change. While over time, all design decisions are liable to be altered, some are more likely than others. These decisions that are likely to change become the "secret" of the module. Heninger and Shore [12] have identified some of the more common "secrets" in a data processing system to be:

Logical data base structure.

Algorithms used in performing the various tasks of the module.

Data storage device physical representation.

Input mediums (cards, tape), record fields, sequence.

Output devices (printers, tape, cards) .

Operating System interfaces.

Software functions as seen by the user.

Hardware configuration characteristics.

Parnas argues that there should be a single "module" that possesses all of the portions of the software system that might be affected by one of these changes. A key distinction between what Parnas describes as a module and the more classic concept is that he distinguishes between design modules and implementation modules. A subroutine or program (at execution) may well be "an assembled collection of code from various (design) modules." [58]

Each module is defined in terms of the interface between itself and the other modules. These interfaces are, in effect, the only assumption that the other module and their implementators are allowed to make. Thus, the interfaces must be extremely specific in describing the acceptable range of inputs and outputs that the module will require. On the other hand, the interfaces should contain no information that describes how the module transforms the inputs into the output format required. This is what is meant by "information hiding".

This concept of information hiding modules is that each module is to hide an assumption about the solution that is likely to change. By hiding how the module performs its required transformation, three major benefits are derived. First, the personnel who are responsible for implementing the various modules are prevented from creating unnecessary interdependencies between the modules. When the changes then occur, the effect of the change is limited to a single module. Secondly, by not specifying how the module is to be implemented, it gives the programmer the freedom to be creative and to determine the best way to accomplish the module's task. Lastly, this method requires an explicit and precise statement of the interfaces between the modules and forces the designer to analyze and state those secrets that are most likely to change.

While the concept of information hiding is often discussed and endorsed as a theoretically sound idea by authors in the field of software engineering, not all are convinced of its practicality.

Yourdon [57] for example, while recognizing the usefulness of the concept of isolating the shared assumptions between modules as a means of reducing the level of coupling between them, offers two criticisms of this approach. First, he believes that there is no procedure offered with respect to how to apply the criteria. Secondly, the critical problem of translating the design modules and interfaces into programmable, interconnected modules is not addressed by Parnas in his description of this technique of modularization.

F. L. Brooks [52], while commending the Parnas approach for the idea of interfaces that are completely and precisely defined, has labeled the dependence on its perfect accomplishment a "receipe for disaster."

In an attempt to determine the feasibility of this approach as well as to provide a useful model for future developers, Parnas, Heninger and others are currently involved in a project sponsored by the Naval Research Laboratory and the Naval Weapons Center to redesign and build the operational flight program for the A-7E aircraft. Described in detail in References [60, 61, 62], the redesigned program will be functionally identical to the existing A-7E OFP so that direct comparison between the two can be made in terms of development time and cost, resource utilization, and for ease of change.

Just as there is no agreement as to the correct criteria for decomposition of large systems into modules, there is also no consensus as to what constitutes the optimal size of a module. Baker gives a commonly used limit of 50-60 statements; equivalent to the number of printed lines that will fit on a single page of computer printout [63]. Constantine suggests a range of between 100-200 statements [57]. Yourdon, in Reference 21 reports of an Air Force Project whose project manager, concerned about the number of modules due to the estimated size of the project, imposed a standard of no more than 500 COBOL statements per module. While the last number appears to be somewhat high, the point is that each module should be easy to understand and remember. It should be noted, however, that a large number of small modules in a program can result in an increase in the overall complexity of the system.

The key of both the functional approach and Parnas' information hiding modules is to create software that is more readily designed, understood, and changed by reducing large, complex systems into smaller, independent and more manageable components. An additional benefit of modularity is that by dividing the problem into smaller pieces, the assignment of personnel to design and implement the system can be made on the basis of the division. This may well have a positive effect on the time and cost of development by allowing independent groups or individuals to work simultaneously on the individual parts rather than attempting to manage a large, monolithic project.

2. Abstraction

Like modularity, abstraction is a very pervasive principle in software engineering. The essence of abstraction is to extract the essential properties of a system while omitting or postponing the consideration of non-essential details. It is vital to the development of software that is machine and implementation independent. The purpose of abstraction is similar to information hiding in that it requires making visible only those properties that needed to describe a module in terms of its function and interfaces with other modules within the system. Abstraction, however, differs from information hiding in that abstraction omits unnecessary details whereas information hiding first must identify and then hide details that constitute the secret of the module.

The design of computer system that identify the functions required of the system without regard to whether they are to be implemented in the hardware of software, i.e. the "virtual" machine concept, is one example of the use of abstraction. There is also an abstraction involved in using a tool or device to accomplish a goal while disregarding the reason it functions as it does. For example, an individual may use a mathematical theorem to prove a program's correctness without considering how the theorem was originally proved.

Parnas in describing how to specify the interfaces, or assumptions, between modules of a program uses the phrase abstract specification. His contention is that while the specification must be precise, it is abstract if it states the requirements to be met without referring to a theoretical or real implementation [30]. This abstract interface serves as a model of the real interface while omitting unnecessary details regarding its implementation. Thus, if the specfication is not precise, the problem solved may not be exactly that whose solution was needed. Finally, the interpretation of the specification used in verifying the correctness of the design decisions may vary from the interpretation made by the implementor in building the module. In fact, as stated earlier, Parnas and others are attempting to provide mathematical notation schemes to describe the specification due to the very imprecision of natural languages, such as English.

Abstraction is, then, a means of modeling several possible solutions to a problem. It is a tool for removing unnecessary detail. In describing the interfaces between the modules of the A-7E program, Parnas defines what he terms an "abstract interface" [30, 62]. An abstract interface is a precise, formally specified set of assumptions regarding the information passed between modules. It is a model of all the actual interfaces in that program. The implementation of this model will be independent of the particular hardware configuration. The crucial idea is that the abstract interface precisely describes the effects of the module on the rest of the system without restricting the programmer in how to effect the implementation. What is true of the actual interfaces must also be true of the model. Otherwise, the specification is not a valid representation of the design. However, by omitting the details regarding the particular device the software is to be operated on, the use of this concept creates designs that are machine independent. By insisting on abstraction, it allows multiple versions of the same interface to be implemented from the same design thereby contributing to the portability and reusability of the design.

3. Hierarchical Design Approaches

There is a great deal of disagreement about the relative benefits and disadvantages of various hierarchical structures within the field of software engineering. Non-trivial hierarchical structures, by their very nature, imply restrictions that are placed on the designer. A hierarchical structure is a structure with no loops in its relationships and consists of two components, its parts and the relationship between the parts. The two most common design approaches for developing hierarchial structured software systems are top-down and the bottom-up.

The phrase top-down is probably one of the most commonly used terms in computing. Top-down design or, as it is referred to by Dijkstra and Wirth [29, 64], "stepwise refinement," involves in a very general sense first specifying the system in its broadest terms and in a stepwise, iterative method, refining the structure by filling in the details. This refinement entails filling in the details of the highest level of the structure until it is completely defined before progressing to the next lower level. At each stem, major functions or tasks to be accomplished, along with the inputs, outputs, interfaces and constraints are identified and incorporated into a design decision. This decision is described in terms of a functional specification formulated in some suitable notation. At each level all of the details are filled in and possible candidates for further refinement are identified.

Although the order in which to make the decisions vary from author to author, Wirth [64] has suggested the following guidelines. Decompose decisions as much as possible to separate aspects which may initially appear related. Defer those decisions which concern details of data representation as long as possible. Base the design decisions upon such criteria as efficiency, storage economy, clarity, and consistency of structure. When considering a particular design decision, alternative approaches should be considered.

Dijkstra [29] offers two additional guidelines. The designer should attempt to make the "easiest decisions first" and compose the program incrementally, deciding as little as possible with each step. As both authors point out, this process is not a single pass approach but rather where each set of decisions are considered in terms of the layer above.

A problem with the use of the phrase "top-down" in conjunction with design is that many authors tend to intertwine top-down design with top-down coding and top-down testing. Design is a method of attacking a problem looking for a general solution. Coding, or programming, is the means of turning the design into a machine executable form. Testing is the means by which the code can be compared to the specifications to ensure that it meets the criteria set forth in the user requirements. It is perfectly possible, and, in some cases, desirable to use top-down design in conjunction with bottom-up coding and/or testing.

In a general sense, all design is to a large extent top-down in that the human mind generally first recognizes a problem in the large sense and then attacks that problem by breaking it down into smaller, more manageable parts. However once an overall solution has been determined, a bottom-up approach can be utilized to fill in the details. McClure [65] characterizes the bottom-up approach as a well-formed conceptual level (bottom level) that is a set of well-defined interrelated concepts that may be combined to make up more elaborate concepts. The bottom-up approach starts at the level where the software interfaces with the machine or operating system and tries to take advantage of particularly desirable attributes of the machine in making design decisions. One problem with this approach is that it tends to lead to software that is machine dependent and, thus lacks portability and reusability except on machines of a similar architecture. Another potential problem is that if the higher levels are not clearly defined, the bottom-up approach may well lead to highly complex systems that do not ever reach the top or do so only with great difficulty. However, this approach can be used as a means of solving design problems. McClure [65] reports that even Dijkstra, considered by many to be the "father" of the top-down approach, utilized a design approach that was primarily bottom-up in nature in designing the "THE" operating system [66] because he was chiefly influenced by the existing hardware on which the system was to run.

Yourdon [21] also considers that there are a number of situations where it is reasonable to combine the bottom-up with top-down design. Two examples of these situations that he provides are:

1. The designer is aware that a number of utility routines already exist and he tries (either consciously or unconsciously) to adapt his design to make use of them.

2. At a relatively early stage in the design of his program the designer anticipates that certain common or general-purpose routines will be required by several different portions of the program, such as error routines, editing routines, input-output routines and table look-up routines.

He contends that if done properly, a small amount of bottom-up design can be practical although he cautions against getting in a situation where the top portions have to be "bent" in order to make it compatible with the bottom half. This bending can result in software that fails to meet the initial user requirements.

Although recognizing some of the advantages of using a bottom-up approach, this thesis endorses the use of a top-down design approach as the preferred method for designing software systems that are less complex, more understandable, and more likely to meet the requirements of the user.

4. Uniformity

The concept of uniformity, which Ross et. al. [23] define as the "lack of inconsistencies and unnecessary differences" is also an important principle of software engineering. Uniformity implies a consistency in the means of identifying and recording the decisions made during the design, implementation, and operation and maintenance phases of software development. An example of an inconsistency would be to use two different data variable names, one in the design documentation and another within the program itself.

The notion of uniformity also implies the concept of traceability, a characteristic of software that is vital to maintainability. Schneidewind defines traceability as "the ability to identify the technical information that pertains to a software error which has been detected during the maintenance phase and thereby trace the error to the applicable design specifications and user requirement statements." [67] The notion of traceability is integral to the maintenance of software in that it is a necessary attribute of software that assists the maintenance programmer in discovering the cause of errors as well as allows him to make corrections that are consistent with the original design decisions and user requirements.

Various schemes have been developed to help achieve traceability and uniformity. They include adoption of variable and module naming conventions; the use of numbering schemes in both the source code listings and the associated documentation that permit an individual module to be traced back through the entire design process; the use of data dictionaries to define all of the variable names in a precise notation; and the use of graphics to show the overall structure and detailed data flows of a software system. All of these concepts will be developed further in the next chapter.

The concepts of uniformity and traceability, just as the other principles discussed previously, must be considered during the design phase. Documentation as well as the actual code must be designed to ensure that these principles are followed. As Ross asserts [23], a notation scheme that does not allow for uniformity should not be used.

5. <u>Completeness</u>

The notion of completeness is that the user requirements and design specifications must be exhaustively detailed and documented prior to attempting to build the system. The GAO study on maintenance practices [5] found that modifications accounted for about half of total maintenance workload. This estimation concurs with the findings of Lientz and Swanson reported earlier. While recognizing that some modifications occur as a result of adaptation of the software to changing user requirements, others occur because the user needs were not properly identified in the first place. A large number of respondents to the GAO study (171 of 409) indicated that better definition of the user requirements would be the "single most beneficial type of effort to reduce software maintenance [5]."

The user requirement definition forms the basis of all software development. Particularly in those cases where the software is to be developed by a group other than the acquiring agency, it is vital to ensure that the requirements are detailed as completely as possible. In addition to being complete, there are two other considerations in the develcpment of user requirement specifications. First, they should be functionally oriented; i.e., they should describe the system in terms of what it should do, not how it should be implemented. Secondly, they should be precise. They should contain measurable and quantifiable attributes of the system.

The reason for this should be self-evident. If the testing of the software is to be done on the basis of the specifications, then they must contain criteria for the measurement of how well the software met the requirements. By insisting on specifications that are complete, functional and precise, it will help ensure that the system that is built fully meets the needs of the user. It should be noted that some authors, particularly those that have had dealings with the federal government, caution against overspecification. Overspecification occurs primarily when the specifications begin to tell the developer how to design and implement the system rather than what the system is to do. Historically, however, the problem has been more of underspecification rather than the reverse. Reference [61], which details the software requirements for the A-7E Operational Flight Program project is one model of precise, testable specifications that are both complete and concise.

6. Confirmability

As was implied in the last section, one of the most important principles of software engineering is that of confirmability. One aspect of confirmability is testability. There exists numerous books and articles describing various approaches to testing. One such source is Reference [21] which details and compares the top-down and bottom-up philosophies of approaches to testing. An excellent overview of various automated tools for test is contained in Glass' <u>Software Reliability Guidebook</u> [32] and in Ramamoothy and Ho's "Testing Large Software with Automated Software Evaluation Systems" [68].

One type of testing not often addressed which has a direct impact on maintenance practices is regression testing. Regression testing is a method of detecting errors in changes or spawned by changes made during software maintenance [32]. The purpose of regression testing is two-fold. One purpose is to ensure that the problem to be corrected is, in fact, resolved. Secondly, regression testing is to ensure that no new errors are created with respect to the other modules that interface with the changed module. If acceptance testing has been done properly, the test data used for that test can form the nucleus for the regression testing. This implies that the original test data, including the test plan, must be retained. As new modules are inserted into a program, new test cases may be required to be developed to cover possible cases that did not exist during the intial testing phase. Just as software systems are designed, so must test cases be designed to ensure the critical areas of the system is checked. It is important to note that complete testing of all possible combinations of paths through a system is, except for the most trival cases, impracticable given current technology. Boehm [5] has described a reasonably simple control structure that would, at current processor speeds, require over 2000 years to exhaustively test each possible flow path.

As Schneidewind [67] points out, there is no current requirement for regression testing in either Military Standard 1679 (Navy) Weapons System Software Development, or Weapons Specification, WS 8506. Another issue is that neither of these documents address any requirement for directly testing software to determine its maintainability. Chapter VI examines two possible approaches to testing for this vital characteristic.

Another issue in confirmability is the verification and validation of intermediate design decisions. Validation is the determination of the "correctness of the software produced, including documentation, with respect to the users needs or requirements. Validation is usually accomplished by verifying each stage of the software lifecycle." [69]

Verification is the "demonstration of consistency, completeness and correctness of the software at each state and between stages of the development lifecycle." [69]

Both of these concepts differ from testing in the usual sense in that testing implies the examination of a program by executing it using sample data sets, validation and verification consists of determining that each stage or iteration of the development process is consistent with both the decisions made at the preceeding phase as well as the fulfilling the user's requirements and program specifications. Techniques for validation and verfication are largely manual in nature and consist of such techniques as structured walk-throughs [21, 57], formal design reviews, and peer code reviews [70].

Military Standard 1521 (USAF), "Technical Reviews and Audits for System, Equipment and Computer Programs," 1 June 1976, levies the requirements for design reviews to be conducted during software development. Under this standard the following technical reviews and audits are required:

System Requirements Review	(SRR)
Systems Design Review	(SDR)
Preliminary Design Review	(PDR)
Critical Design Review	(CDR)
Functional Configuration Audit	(FCA)
Physical Configuration Audit	(PCA)
Formal Qualification Review	(FQR)

Detailed definitions and specific requirements for each of the reviews and audits are contained in the standard. While it should be noted that this standard fails to list requirements specifically considering the optimization of the maintainability of the software product, a companion guidebook [71] provides checklists of maintenance considerations for use in conjunction with the reviews and audits.

IV. SOFTWARE ENGINEERING TOOLS AND TECHNIQUES

A. THE SOFTWARE LIFE CYCLE

A major thrust of this thesis so far has been that the consideration of quality must be done from the beginning of the development process rather than left until after the design of the system is complete. Prior to discussing some of the tools and techniques that have been championed in the literature as helpful in the development of quality software, it is necessary to examine all of the phases of the software life cycle through the phase where maintenance occurs. Unlike the hardware life cycle, which is well-established in the literature, there exists no universally accepted model of the software life cycle, with well-defined boundaries and interrelationships. Therefore, several models will be presented in order to provide a broader understanding.

Glass [32] divides the life cycle of software into five distinct phases:

Requirements/Specifications

Design

Implementation

Checkout

Maintenance

The requirements/specifications phase is the phase where the problem is being understood and initially defined. He set forth in the specifications. Checkout is the process of seeking programming errors, conceptual errors in the design, questioning requirements and "putting the final polish" on the software. He identifies the greatest hazard of this phase to be impatience and cautions against inadequate testing and examining before delivering the program to the user.

The final phase is maintenance which he defines as "the process of being responsive to the user's needs, fixing errors, making user-specified modifications and honing the program to be more useful." [32] He contends that maintenance, while being "unglamourous" and the "Siberia" of programming, where new, inexperienced programmers are trained before moving up to design, may be the most important activity in terms of user satisfaction. The greatest hazard to maintenance is "ineptitude" which can undo all of the good of the previous phases while "turning a finely tuned Stradivarius into high-quality fireplace wood by a ham-handed maintainer." [32]

While one advantage of this model of the software life cycle is that it recognizes the effect of good maintenance practices on software quality as well as the need to consider early the characteristics of quality to be emphasized, it does little to illustrate the iterative nature of the software life cycle.

Boehm [4] offers a more detailed definition of the life cycle of software, as illustrated in Figure 4-1. The article

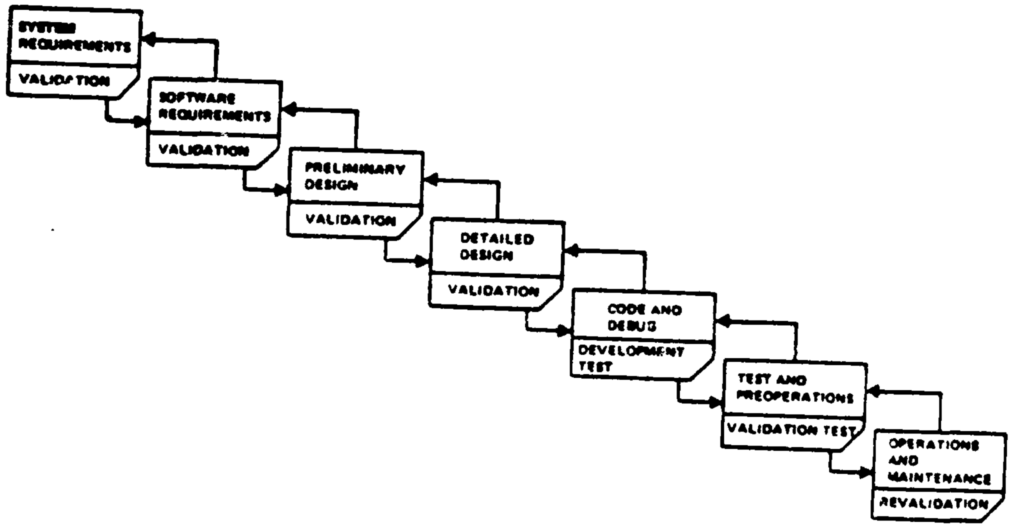


Figure 4-1. Boehm's Software Life Cycle Model [4].

emphasizes that during this phase, the concern should be to clearly define the problem rather than attempt to devise a solution. He states that the greatest hazard during this phase "...is the temptation to define a solution to part of the problem, ignoring the hard parts or those that are ill-defined." [32]. He continues that

> "Succumbing to this temptation leads to inadequate design and implementation, which in turn, leads to revised requirements and modification. Thus well-thought through requirements and specifications, which are the primary output of the requirements/specification phase...are vital to both the quality and reliability of the software they define." [32]

The second phase is the design phase which translates the problem into a conceptual solution. While recognizing a growing concern that traditional design approches often stop too soon, leading to inadequate solutions and a high number of design errors, Glass also cautions against "grinding a design deeply into the nitty-gritty implementation details." [32] He contends that this effort not only wastes time and money but also is, at best, a replication of the implementation process.

Implementation, the third phase in his model, translates the conceptual solution of the design phase into computer-processable form. The greatest risk of this phase is carelessness which, in his words, "...can turn a Stradivarius into a K-Mart toy." [32]

The fourth phase, checkout, is the process of examining and testing the software to see if it meets the standards in which this model was presented has become to be considered a classic in the field of software engineering. It is a summary of the field with a description of many of the tools and techniques and is particularly noteworthy for its extensive bibliography. Yourdon, in an introduction to the article which is reproduced in Reference [72], has stated:

"If someone said to me, 'I have time to read only one of the papers in your collection,' I would recommend this paper by Barry Boehm - not because it is brilliant (although I think some of Boehm's insights border on brilliance) or because it revolutionized the field (the way that some of Kijkstra's papers did), but simply because it is probably the best overall summary of the software field that I have yet seen published."

This model has several advantages. It focuses attention on the highly iterative nature of software design, indicated by the feed back arrows from each phase to its predecessor. It also highlights the need to validate each phase, including maintenance in terms of the previous decisions made.

There are, however, two disadvantages to this model. It does little to illustrate the desirability of user involvement in defining the requirements for the system. It makes the assumption that the problem has been correctly identified prior to the specification of the overall system requirements. It also fails to emphasize that maintenance activities can require respecification, redesign, reprogramming as well as retesting.

The final model that will be presented is that developed by the Rome Aid Development Center, which is shown in Figure 4-2 [6]. It appears to accurately model the software

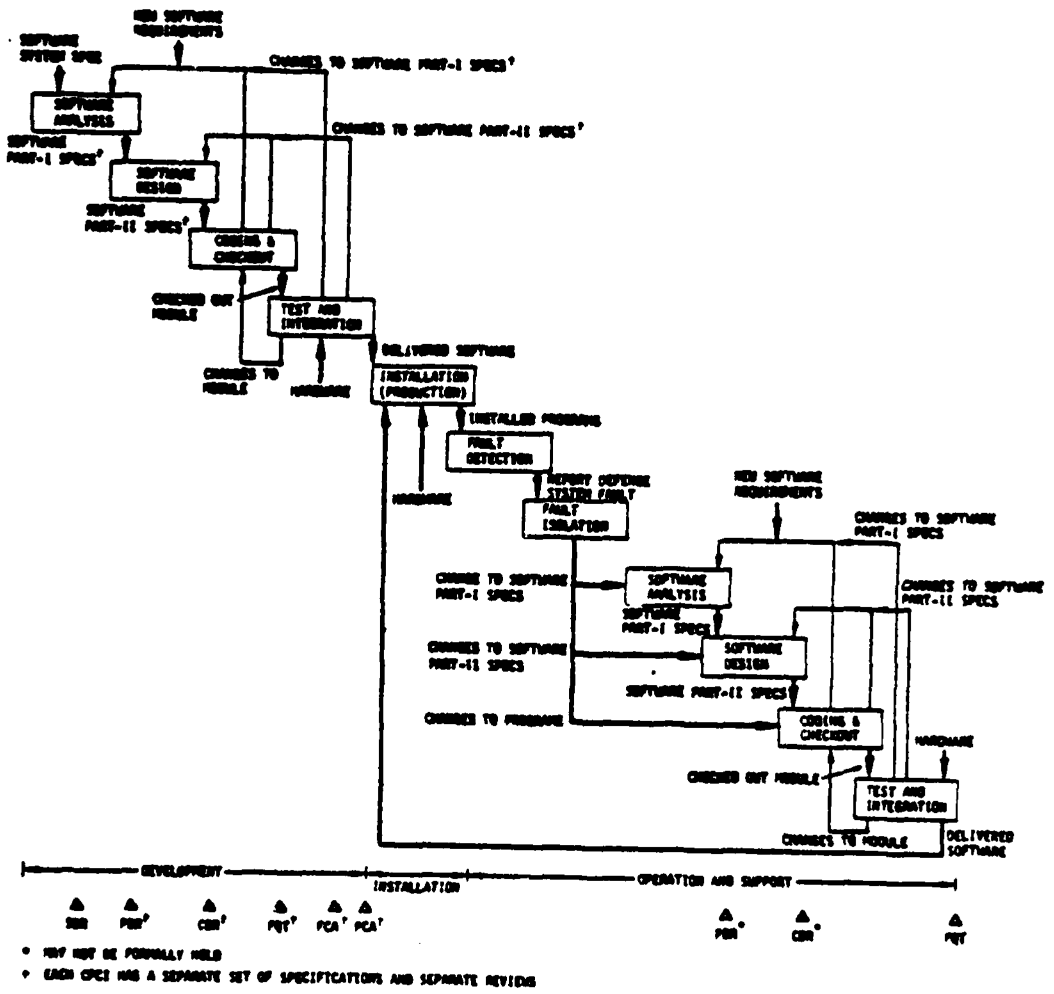


Figure 4-2. RADC Software Life Cycle [6].

life cycle. It shows the process of software development to be highly interactive and iterative in nature as is indicated by the arrows that accomodate new requirements and changes to the specifications. More importantly, it emphasizes the importance of the operation and support phase which divides maintenance into a series of subphases. This division highlights the concept that maintenance activities include the same analysis, design, coding, checkout, test, and integration phases as the initial development process. An additional advantage of this model is that it illustrates the temporal relationship between the software life cycle and the technical reviews and audits described earlier as well as denoting the various baselines established as part of the configuration management process.

Figures 4-3, 4-4 and 4-5 further reenforce the concept that it is both necessary and desirable to consider quality characteristic requirements early in the process of software development. Combining the results of several different studies by various authors, Glass [32] provides a convincing visual argument that serves to further highlight this notion.

Figure 4-3 shows the relative percentage costs of each phase of the software life cycle. There are two major reasons that possibly explain why the earlier phases represent such a relatively small portion of the overall life cycle cost of software. First, it is plausible that too little attention and time have traditionally been spent on ensuring that these

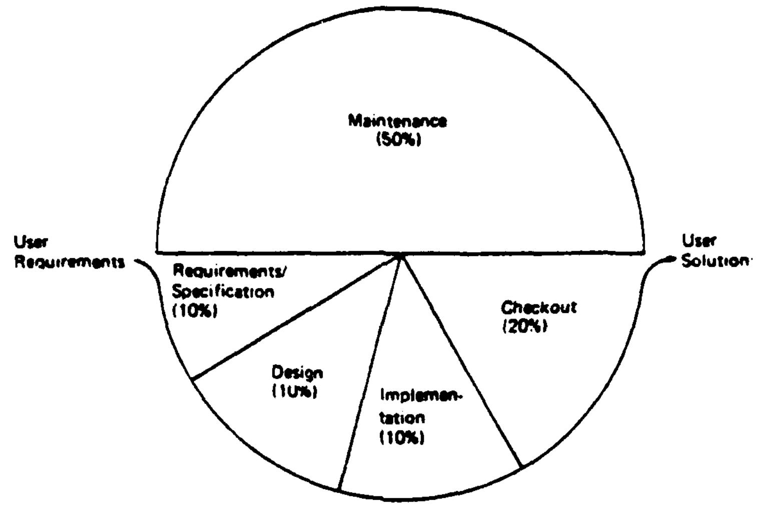


Figure 4-3. Software Life Cycle: Costs per Phase [32].

phases are adequately performed and validated. Furthermore, because of this lack of attention to proper validation, the maintenance costs are increased due to the need to correct errors that were missed previously. A second possible explanation is that the latter phases are where traditionally more manpower is added, due to both the labor-intensive nature of the activities as well as a means of making up time for previous schedule slippages. This latter practice has led to Brooks postulating his now famous law which states that "Adding manpower to a late software project makes it later." [52] His contention, which has been partially validated in Reference [73], is that, due to the added time needed for communication between personnel familiar with the project and those newly-hired personnel added to increase productivity in order to meet the project deadline, the average productivity is redeced further until the new personnel have been trained.

Figure 4-4 shows the percentage of errors as a function of the phase in which they occur. It further emphasizes that there is a need for careful design validation and verification techniques as a means of reducing corrective maintenance costs.

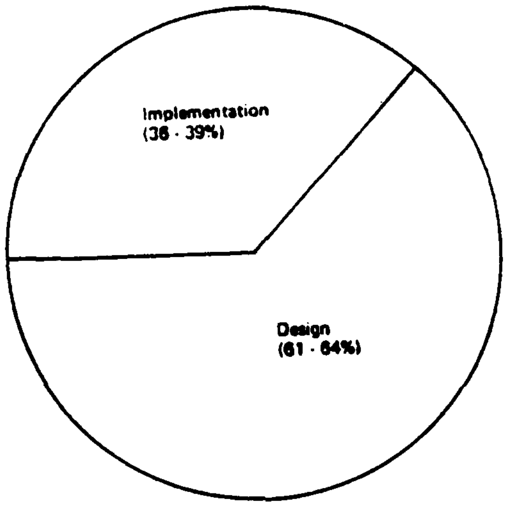


Figure 4-4. Software Life Cycle: Error Sources per Phase [32].

Figure 4-5 graphically illustrates the relative cost to correct an error as a function of the phase in which it was discovered. The reason it would appear less expensive to correct errors in the earlier phases of development is that fewer binding and interrelated decisions have been made. Therefore, making a change in one area of the program or project has a lesser effect on the rest of the systemic decisions that have been made.

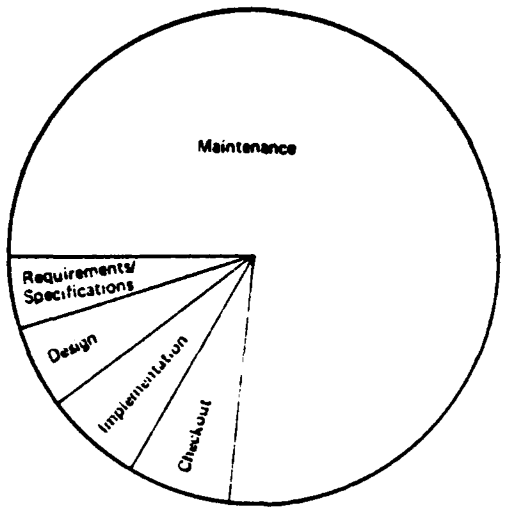


Figure 4-5. Software Life Cycle: Per Error Fix Per Phase.

Clearly, it is in the best interest of the project manager to consider early the requirements for quality and maintainability. As illustrated above, maintenance is affected by decisions made throughout the life cycle. Furthermore, it has been shown that the total life cycle costs may be reduced by this early consideration. It must be emphasized that although maintenance is chronologically last in the software life cycle, it must be properly considered and planned for early in the development process. The next sections of this chapter examine various design and management techniques that have been offered in the literature as a means oι achieving quality. Although it has been shown that the software life cycle consists of many interrelated phases, for the purpose of illustration the tools and techniques will be presented under the following headings: requirement analysis and specification, design and management tools and techniques. Also, a section will be devoted to examining means of graphically representing the design decisions that are made. The emphasis will be on presenting an overview of the various methodologies and no claim of completeness regarding the discussion of any of the individual topics is made. However, references of each of the tools or techniques are provided for the interested reader.

B. REQUIREMENT ANALYSIS AND SPECIFICATION TOOLS

The process of analyzing the needs of an organization to determine the requirements of and constraints on a software system has traditionally been called systems analysis. Although much of the work done during this phase is done by personnel other than software engineers and is thus beyond the scope of this thesis, Jensen and Tonies highlight the need for the involvement of the software engineer in the analysis of the requirements. Although, in their words,

> "They (software engineers) do not play a lead role in systems design, the software engineers are instrumental in bringing all of the elements of the system together in the final systsms product and making them operate together. Therefore, the software engineers play a major synthesis role in the design process." [17]

1. Automated Tools

Although the process of requirements analysis is primarily a mental activity that requires the analyst to combine the needs of the users with various technological and economic constraints to produce a clear and precise definition of the system, there have been attempts to automate the process in order to provide a means of specifying the requirements in such a manner as to ensure their completeness and consistency as well as facilitating ease of change.

A problem statement language (PSL) and problem statement analyzer (PSA) are two tools developed by Teichroew and others at the University of Michigan to aid in the systems design process. [74] The PSL is based on the definition of objects and their relationships, providing for more than twenty types of objects and over fifty relationships. The PSA, a supporting software system, accepts PSL as input and is used to maintain a data base of current specifications. It produces a number of reports and graphics that describe internal and external data flows, management information and other systems data. One feature of the PSA is that it checks for inconsistencies in naming of variables and data flows and is useful in identifying areas that require further analysis. Another management feature is that it tracks changes to specifications by date, the person making the change, and the person who authorized it. This information allows the manager to keep track of the status of the analysis as well as providing a history of the design decisions that were made or changed.

SREM or Software Requirements Engineering Methodology is a system that was developed for the U.S. Army Ballistic Missile Defense Advanced Technology Center by TRW for use in the analysis and specification of requirements for large, real-time weapons systems [75, 76]. It utilizes a PSL-like language called RSL, or Requirements Statement Language, along with a number of computer programs to support the creation and checking of requirement specification documentation.

The primary goal of both of these efforts is to develop complete, precise and consistent specifications that are easily understood by both the user and the developer. It also facilitates the modification of the specifications by storing them in a data base with programs designed to automatically generate specifications as well as provide a means of tracking progress and changes made in the system.

Reference [77] has examined the use of these systems in the Department of the Navy. PSL/PSA, while promoted in the literature as primarily a business oriented system, has been, along with SREM, utilized primarily for tactical software development. The author has identified several problems associated with their use in DoN projects. Both require large supporting computers to hold both the data bases and support software. SREM, in particular, is a highly machine dependent system due to its memory mapping technique. It operates with approximately 60,000 lines of PASCAL code and can be operated on a limited number of computers; namely the Texas Instruments Advanced Scientific (ASC) computer and certain models of the CDC 6600. Petrie [77] reports that work is underway to also make SREM compatible with Digital Equipment Corporation's VAX-11. PSL/PSA, which is written in FORTRAN, is somewhat more portable but also requires a large memory capacity. Another weakness is that users have found that due to the syntactical characteristics, there is considerable training required before they can use these tools effectively.

A third major problem with these systems, and one that has a significant impact on potential DoD users, is that the outputs produced by these systems do not meet current standards for software specification documentation for either tactical or non-tactical software as delineated in References [78, 79], and [80]. Research efforts have been undertaken to develop programs that will translate the current outputs into an acceptable format. Glass [32] cautions against the use of these systems except where the users have had prior experience. Furthermore, due to the cost of the support equipment and software, he feels that they should be used only on projects of a large scale. Yet, clearly, the potential impact of automated tools such as these on the production of software that is complete and reliable as well as more easily modified is great. Further research and development to make them more portable and user-friendly is both needed and justified.

2. Structured Analysis and System Specification

This technique, described in detail in Reference [81], is intended for use with Structured Design [21, 57] as a means of tying together the requirement analysis and specification and design phases of software development. Its proponents claim that it is a top-down, partitioned, graphic way of analyzing the user's requirements that result in the production of a structured specification.

DeMarco [81, 82] notes the following five advantages of this approach:

1. It is graphic, made up mostly of diagrams.

2. It is partitioned, not a single specification but a network of interconnected mini-specifications that make the reading and understanding of each part as well as the system as a whole, easier.

3. It is top-down, presented in a hierarchial fashion with a smooth progression from the most abstract upper progression from the most abstract upper level to the most detailed bottom level.

4. It is maintainable; a specification that can be updated to reflect change in the requirements.

5. It is a logical model of the system-to-be. The user can work with the model to perfect his vision of business operations as they will be with the new system.

The purpose of this methodology is to identify and track the flow of data through a system noting the processing required to transform the input data into an output format that is acceptable to the user. The goal of this approach is to produce "structured specifications" that consist of the following elements: Data Flow Diagrams for each level of modules in the system, a Data Dictionary, and a "structured" English or decision table representation of the processing logic at the primitive level in what is termed a "mini-spec".

a. Data Flow Diagrams

A Data Flow Diagram or DFD is a network representation of the system. It portrays the system in term of its components with all of the interfaces among the various components identified. There are four symbols used in the notational scheme to represent the components of the system. A circle or "bubble" represent the process being performed on the data to transform it into the format required for output. Data flows are represented by arrows with a unique identifying name for the data flow above the arrow. Data sources and sinks, i.e., the beginning and end destination of the newtwork, are denoted by a box with the name contained therein. Files that are required by the system to hold the data either temporarily or permanently, are represented by straight lines with the file name next to it.

The concept of a leveled DFD, as protrayed in Figure 4-6, is a result of top-down analysis. The top level or "context diagram" is a departitioned version of the entire system that shows only the net inputs and outputs of the system. Its only purpose is to delineate the scope of the study. Each process identified is numbered with the processes identified as the highest level assigned the number zero.

[illegible] 423 SOFTWARE ENGINEERING BASICS: A PRIMER FOR THE PROJECT MANAGER (U)
STEVEN PATRICK ARTZER, ET AL NAVAL POSTGRADUATE SCHOOL, MONTEREY,
UNCLASSIFIED CA JUN 82

2 OF 3
ADA
120423

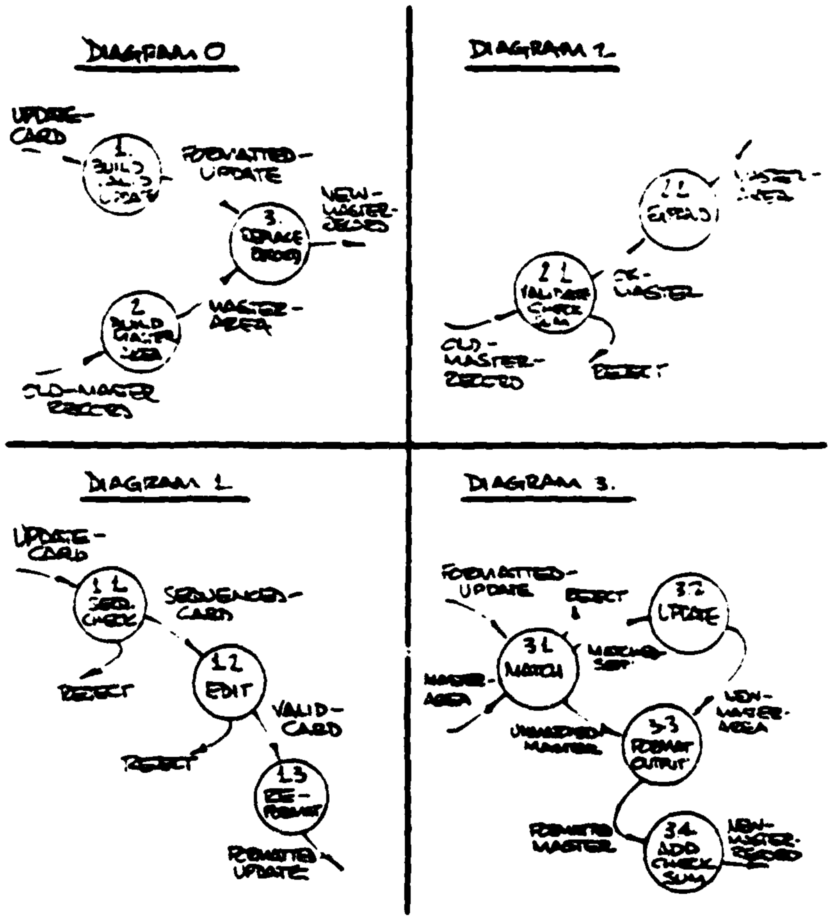


Figure 4-6. Example Data Flow Diagram [81].

All components associated with a particular level are identified prior to beginning work on the next lower level. Each process is examined and a decision is made as to whether it represents a single function. If it does not, subsequent middle level diagrams are constructed with the numbers representing the subdivisions of the upper level process becoming an extension of the upper level process number. Thus, diagram 1 of Figure 4-6 is a further refinement of the process 1 from diagram 0. If, for example, bubble 1.1 was determined to need furhter expansion and found to consist of 3 subparts, the resulting diagram would be numbered 1.1 and each subpart 1.1.1, 1.1.2, and 1.1.3 respectively. Each diagram is designed to fit on and 8 x 11 inch sheet of paper. The entire package of diagrams would consist of all of the ones necessary to trace the system from its highest level of abstraction to the lowest level primitives. If an error or change was required to any single process, the effect of the change would be isolated to only that diagram and those of its lower level derivatives. The numbering convention also aids in tracing the effect of a change on the lower levels. DeMarco gives the following summary of leveling conventions:

"1. To see the detail of a given bubble, look at the diagram with the same number.

2. Inputs are balanced between parent and child - data flows into and out of the parent bubble are equivalent to the data flows into and out of the child diagram.

3. At any given level, only files and data flows that are interfaces among the DFD elements are shown. Files and data flows that are only relevant to the inside of some process are concealed.

4. At the first level where a file is shown, all references to it must be shown." [81]

b. Data Dictionaries

The second component of the structured specification is the Data Dictionary. It is the "set of definitions of data flows, data elements, files, data bases, and processes referred to in a leveled DFD set." [81] There is a limited set of symbolic relational operators used in defining each of the elements in the set of DFDs: equivalency, and either-or, iterations of, and optional. Each unique element is identified as to its composition, values and meanings, or process description. Each entry in the dictionary is also identified by the diagram number on which it first appears. Depending on the size of the project, this dictionary can be maintained either manually or with the use of a text editor processor. DeMarco recommends using the PSL/PSA automatic system of Teichroew's [74] described earlier as a means of automating the process of building and maintaining the data dictionary. Additionally, this automated tool can be used to generate the DFD diagrams themselves. Thus, the diagrams, as well as the dictionary itself, can be changed and regenerated as new requirements are identified.

The final component of the structured specification is the "mini-spec", which is a statement of the processing required of each functional module of the system. There exists one mini-spec for each of the lowest level primitive and can take one of two forms, "structured English" descriptions or decision tables. It should be noted that while each of the lowest level primitive functions consist of the various subdivisions of the upper level task, they do not necessarily reside at the same numbered level due to the fact that some tasks will require more subdivision than others in order to ensure that the bottom level bubbles represent a single function.

c. Structured English

Structured English, or as it is referred to in other literature, program design language (PDL) [83], pseudo-code [32] or metalanguage [5], is a version of natural English that makes use of a limited vocabulary and syntax. The vocabulary consists of imperative verbs, terms defined in the data dictionary and certain reserved words for logic formulation. The syntax is limited in DeMarco's version to the three basic control structures, sequences, iteration (DOWHILE), and condition (IFTHENELSE). Other versions include the additional structures of DOUNTIL and CASE [83]. Figure 4-7, an excerpt from Reference [81], shows the processing logic for a stock reordering algorithm.

Glass [32] notes while program design languages such as Structured English are a non-graphic representation, the use of indentiation to grpup common actions dependent on

Policy for Ordering New Stock

```
FOR EACH New-Stock Request:
1.  Find Authorization-Form  SUCH THAT
    Reference-Number EQUAL TO Request-Number
    OF New-Stock-Request.
2.  If NO MATCH discard New-Stock-Request
    OTHERWISE:
    Write Purchase-Order For Ordered-Item.
    Select Supplier FOR WHICH Ordered-Item
    appears IN Supplier-Catalogue-Entry.
    Copy Supplier-Name and Address
    ONTO Purchase-Order.
    Copy Purchase-Order-Number
    ONTO New-Stock-Request.
    File New-Stock-Request WITH
    Authorization-Form.
```

Figure 4-7. Structured English Example [81].

a higher level of control and the limited syntax make it more readable and graphic-like. Furthermore, by restricting these languages to the same control structure as is used in structured programming, it makes it easier to translate the design at implementation.

Some controversy has risen around the degree of formality that should be used in specifying a PDL. DeMarco, for example, claims that his form of Structured English is not a rigorous specification language that allows the analyst to effectively code the requirement. He cautions against introducing such rigor into the analysis phase since to do so would direct attention away from specifying the system in terms of what it should do to supplying the implementor with detail instructions on how to do it. Advocates of more rigorous formality cite mathematical-like rigor and the ability to use automatic consistency checking programs as advantages of the formality. Caine and Gordon [83], for example, have developed the Program Design Language System. It accepts a PDL which uses the constructs of hierarchical structured programs. The system, which has a number of supporting software programs, provides a number of useful summaries which give designers updated versions of the design as well as perform consistency checks to ensure that all of the variables have been defined. This is similar to the other automated tools described earlier. Boehm [5] notes that while the structure and limited syntax make it easy for people familiar with programming techniques to use PDL, it is less easy for non-programmers to understand. Ultimately the advocates of these languages envision it as a means of automatically generating code from the design specifications. This capability would reduce the possibility of introducing errors in translating the design into a computer readable format as well as reduce or eliminate the need for programmers altogether.

d. Decision Tables

The other method of representing process logic in the mini-spec is through the use of decision tables, a technique that has long been recognized as a valuable tool in representing design logic [84, 85]. Decision tables have their greatest applicability in "logic oriented programs that must process a large number of decisions, such as problems where numerous alternatives must be exhaustively considered." [32] Figure 4-8 illustrates the basic form of a decision table.

Condition Stub	Condition Entry
Action Stub	Action Entry

Figure 4-8. Decision Table Structure.

As is shown above, the decision table is divided into four quadrants. The upper left quadrant, called the condition stub, contains all of the possible conditions being considered for a particular decision logic. The condition entry frames the condition stub as so to constitute a Boolean with only two possible states, True or False, or Yes or No. The action stub, in the lower left quadrant, contains the actions that result from the condition tested above. The action entries indicate the desired response to the combination of conditions. A dash, "-" in a box indicates a situation where a particular condition is either irrelevant or is mutually excluded by virtue of two contradicting conditions. An example of the latter would be in an algorithm for determining whether or not to assign an officer to sea duty based on his previous assignment. If one rule was predicated upon the fact that the officer had been at sea for the last four years and another was predicated upon the fact that he had been assigned ashore for the last four years, there could not be a case where the officer would meet both criteria. Finally, a "X" indicates the action to be taken given a particular set of conditions. Figure 4-9 is an example decision table representing a process for approving or disapproving of a loan request.

As reported by Pooch [85], one advantage of decision tables is that it is possible to convert them into compilable source via a preprocessor. Another advantage is

Loan Table	R1	R2	R3	R4
Satisfactory Credit Limit	Y	N	N	N
Favorable Payment History	-	Y	N	N
Special Clearance Obtained	-	-	Y	N
Approved Loan	X	X	X	
Disapproved Loan				X

Figure 4-9. Example Loan Decision Table [85].

that their structure is such that it aids in identifying overlooked situations and logical inconsistencies. There are two potential disadvantages to the use of decision tables. First, possible ambiguity may arise as a result of the "don't care" conditions. Secondly, they are not readily usable in cases where the program logic is not decision-making oriented.

e. Evaluation and Alternatives

The Data Flow Diagrams, Data Dictionary and mini-specs all form the structured specification. Among the advantages claimed by proponents of this methodology, such as DeMarco [81] and Yourdon [21], is that it is a well-known method with a history of successful applications across a wide variety of business organizations. Also, there are courses, texts and even video tape presentations offered as training aids in familiarizing personnel with its use. As DeMarco points out "Most of the ideas have been used piece-meal for years. The advantages of this discipline are substantial. They include a methodological approach to specification, a more usable and maintainable product, and fewer surprises when the system is installed." [81]

Critics of this approach, however, note several weaknesses. It requires four different notations, e.g., Data Flow Diagrams, Structured English, Decision Tables, and Data Dictionary formats, that the analyst and user must become familiar with. Secondly, as Wasserman [34] points out, the number of diagrams can become "unwieldy" if the system in question is very large or requires duplicated bubbles. He also objects that Structured Analysis "...dwells upon the technical aspects without providing management procedures that are essential to th ffective use of such a tool. [34]

There exists another suc l called SADT or Structured Analysis and Design Techniques, a process developed by Ross [86, 87] and copyrighted by Softech, Inc. of Waltham, Ma. It is very similar to Structured Analysis; so similar in fact that DeMarco claims that the major difference is that SADT uses boxes instead of circles to denote processes on its graphics. One other difference is the emphasis of management procedures to be used in conjunction with the modeling aspects of the system. All diagrams go through what Ross describes as an "author-reader cycle." [86] As each level is completed, the analyst/designer reviews the model of the system to-date with the user to ensure that it accurately reflects the user's requirements. It, of course, requires that the user be familiar with the notation used in the diagrams and dictionary. More importantly, it emphasizes both the need for the user to adequately determine his requirements beforehand. It also allows the user to track the progress of the project and allows him to become actively involved in the validation process. The other major management procedure requires that all comments must be received in written form, thereby establishing a effective means of communication between all parties during the analysis phase that is traceable. This communication can be either handwritten or done via a text editor.

Jones [88] objects to the requiring of all communications to be in a written format. He contends that it is not possible to conduct a project of any magnitude on the basis of written communications only. A consequence of this is, in his view, that word processing costs have become the second largest expense of the development phases, exceeded only by debugging costs. However, one of the major portions of the cost to word processing is in the development and maintenance of documentation. As Schneidewind and Kline correctly point out the documentation problem has historically been one of a "lack rather than an abundance of documentation". This would suggest that there is a need for more rather than less documentation. Many of the problems of software maintenance appear to be from inadequate documentation, both in terms of quantity and quality. However, there is still a need for oral communication such as design reviews and walkthroughs.

C. DESIGN METHODOLOGIES

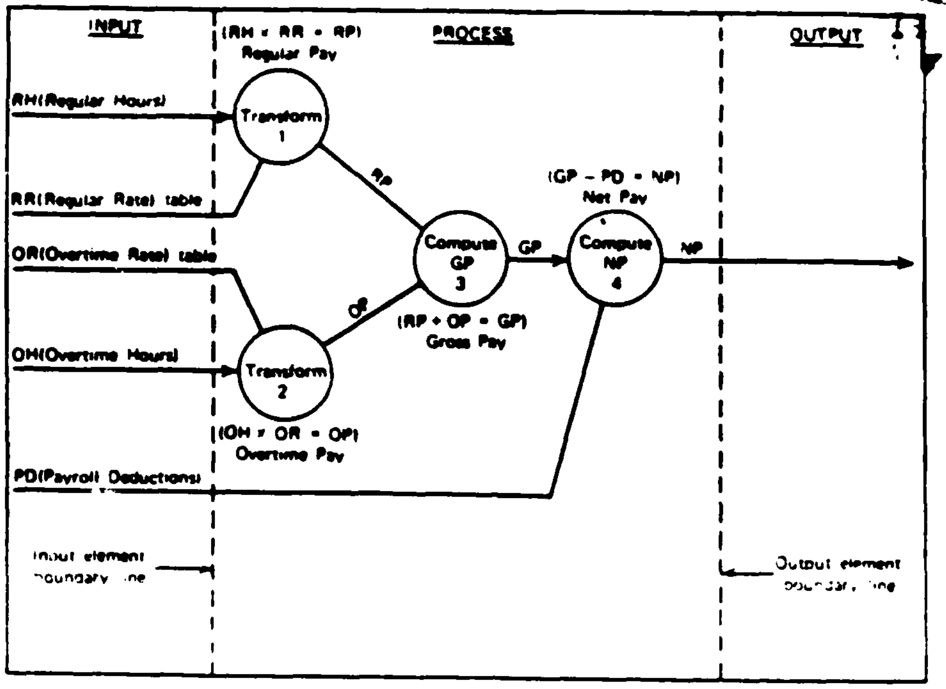
1. Structured Design

There are many touted methodologies for designing software that all claim to be the most "intuitive" method of producing quality software. They can be classified into two major groups: data flow oriented and data structure oriented design.

Under the first catagory, the most publicized method is that known as Structured Design. Several books by Yourdon [21, 57] as well as an article by Constantine, Stevens and Myers [89] all describe this approach in detail. It has been developed to be used as the transitioning tool between Structured Analysis and actual implementation. In fact, articles predating the publication of Reference [81] often combine the two methods.

Structured Design consists of various concepts, measures, analysis techniques, rules of thumb and terminology that aims at transforming the Data Flow Diagram into what is known as a Structure Chart. A structure chart is "a documentation technique for illustrating the modules in a system, and the interconnections between the modules." [57] It represents the actual program modules and identifies the boundaries between modules by identifying the "afferent" (inputs) and "efferent (output) data flows for each module.

Figure 4-10 is example data flow diagram and resulting structure chart for an upper level module that computes net pay as part of a financial accounting or payroll system. Module A represents a control module that requests and receives various information from two I/O devices pulse an internal system table. It passes that information to two transform modules which calculate the gross and net pay.



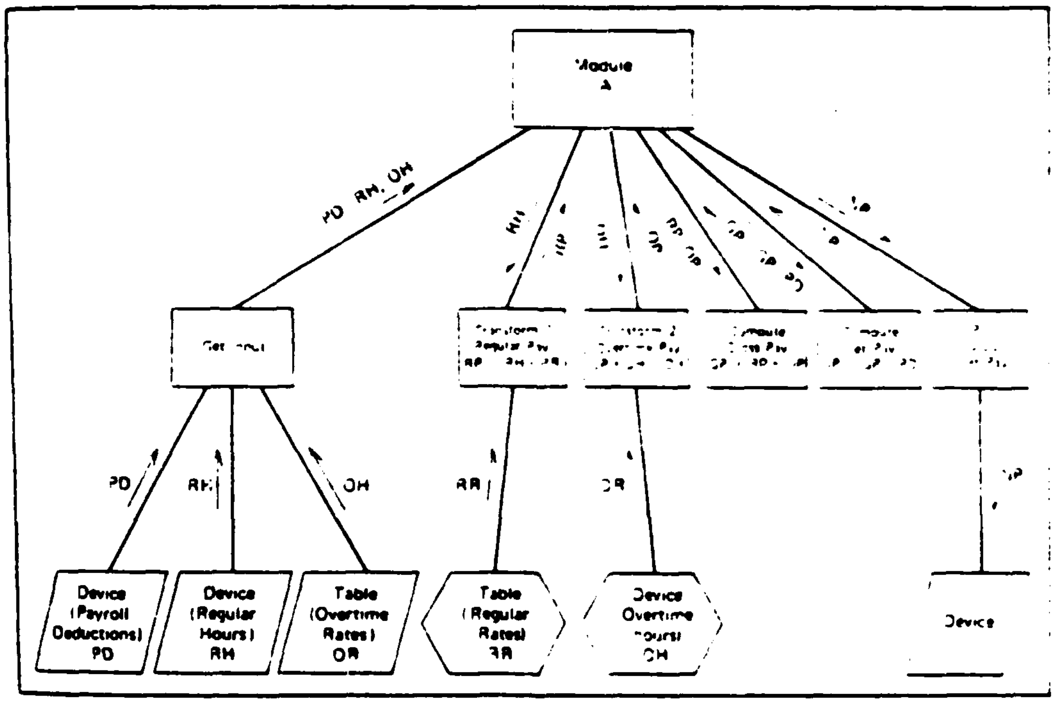


Figure 4-10. Net Pay DFD and Structure Chart [90].

The rightmost branch then outputs the net pay to some periphial device such as a printer. Note that devices are represented in this diagram by parallelograms, internal tables by hexagons, and processes by boxes. The arrows represent the afferent and efferent data flows and are labeled as to the information required by each module or submodule. Detailed descriptions of the data flows would be contained in an accompanying Data Dictionary. The processing logic for each module would also be availabe there. Structured Design uses a strategy called "transform analysis" that translates the Data Flow Diagram into a Structure Chart and determines the shape of the chart. Yourdon defines transform analysis as:

> "A design strategy in which the structure of a system is derived from an analysis of the flow of the data through a system, and of the transformations of the data. [57]

One claimed advantage of this design approach is that by adjusting the boundaries between the inputs, outputs and processes, several alternative structures can be rapidly produced and evaluated. Figure 4-11 shows the same net pay DFD with the boundaries adjusted so that Transforms 1 and 2 are outside the process boundary line. Notice the radically changed appearance of the resulting Structure Chart. All of the calculations other than the actual determination of the net pay are not performed by a new module called "Get transformed data" which is an additional control module in the system. There are now eight vice six processing modules in

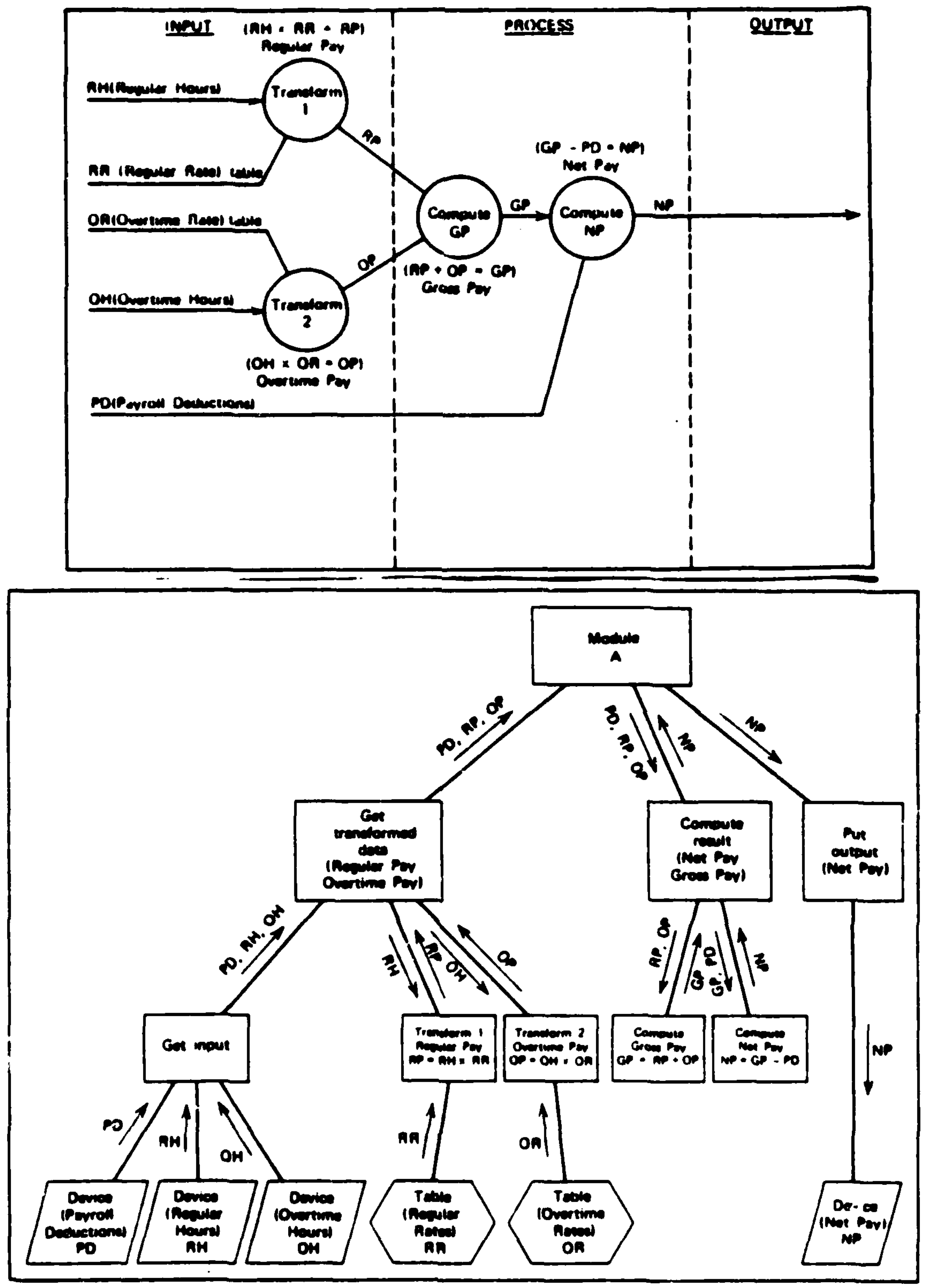

INPUT
PROCESS
OUTPUT
(RH × RR = RP) Regular Pay
RH(Regular Hours)
Transform 1
RR (Regular Rate) table
RP
(GP - PD = NP) Net Pay
Compute GP
GP
Compute NP
NP
OR(Overtime Rate) table
OP
(RP + OP = GP) Gross Pay
OH(Overtime Hours)
Transform 2
(OH × OR = OP) Overtime Pay
PD(Payroll Deductions)
Module A
PD, RP, OP
PD, RP, OP
NP
NP
Get transformed data (Regular Pay Overtime Pay)
Compute result (Net Pay Gross Pay)
Put output (Net Pay)
PD, RH, OH
RH
RP
OH
OP
RP, OP
GP, PD
GP, PD
NP
Get input
Transform 1 Regular Pay RP = RH × RR
Transform 2 Overtime Pay OP = OH × OR
Compute Gross Pay GP = RP + OP
Compute Net Pay NP = GP - PD
NP
PD
RH
OH
RR
OR
Device (Payroll Deductions) PD
Device (Regular Hours) RH
Device (Overtime Hours) OH
Table (Regular Rates) RR
Table (Overtime Rates) OR
Device (Net Pay) NP

Figure 4-11. Net Pay With Altered Boundaries. [90]

the previous figure. Peters and Tripp [91] point out that one of the weaknesses of this ability to move the boundaries at will is that there are no formal guidelines provided in the literature for criteria for moving them. Another weakness is that there are few guidelines as how to verify the accuracy of the data flows. This is due to the lack of review guidelines in the literature describing this technique. However, it is certainly possible to conduct design reviews with this approach as well as any other. Yourdon discusses design walkthroughs in Reference [57]. They are, however, between design teams assigned to portions of a large project. No mention is made of any designer-user interaction.

2. The Jackson Design Method

The most advertised example of a data structure oriented design method is that proposed by Michael Jackson [92, 93]. His method takes the view that the identification of the data structure, i.e., by file, record, field, and element, is the key issue in developing quality software. The Jackson Design Method relies upon three fundamental observations:

1. The program structure should be closely related to the problem structure.

2. For many systems, the problem can be reduced to the creation of a mapping from input structure to the output structure.

3. A design method, in order to gain wide acceptance, should be easily teachable and useable by a large number of average designers.

Jackson determines the structure of the input data and creates a chart that illustrates that structure as a tree-like hierarchy. The top of the tree may be a file, or series of files with records, fields of records and elements of fields cascading from the top to indicate the relationships between the various data parts. He then creates a similar chart for the desired output structure. When they match, the system is essentially designed. In practice, of course, the situation is rarely that simple. Input structures and output structures often do not map together. When this lack of symmetry occurs, it is called a "structure clash". A structure clash can be defined as the existence of multiple sets of data which do not possess a one-to-one correspondence at all levels of the data structures. The normal means of handling such clashes is to define an intermediate structure that through a process transforms the input structure to match the output.

There are several implicit assumptions that this method makes. It assumes that only serial files will be involved. It also assumes that the users know how to structure data so that the specifications supplied clearly define the structure. Since it is designed to be teachable and usable by an average designer, it does not look for optimal designs. Rather, as Wasserman [34] points out, it searches for an acceptable design. Unfortunately, there are few guidelines as what constitutes an acceptable design other than it works. Although relatively new to the United States, this approach has been widely accepted in Europe and the rights to this method have been purchased by INFOTECH Ltd. [34]

3. Summary

While there are numerous other methodologies offered in the literature such as DREAM [94] or Higher Order Software (HOS) [95], there are a few points to be made. While recognizing Glass' contention that "design techniques are probably the least amenable to tools and methodologies and the most desperately in need of them" [32], it is difficult to determine which if any are best.

An experiment conducted by Peters and Tripp [91] examined five different methodologies: Structured Design, HOS, the Jackson Method, Warnier's Logical Construction of Programs and Meta Stepwise Refinement (MSR). While the results were affected by their own past experiences they concluded that none of the methods examined would be an asset in every situation. Assumptions made by each method are just that: things that are taken for granted. Each of the articles cited in this section describe various applications where they were used successfully. The applications, due to their illustrative nature, were generally small in scale as well as chosen to make the author's point. Further research in determining the applicablility of these approaches in large scale projects that assess them in terms of how maintainable and reliable their outputs are, is needed. However no method or tool is designer-proof. The presence of dedicated, technically proficient managers to ensure that the design is meeting its requirements will always be necessary. In the final analysis, Peters' and Tripp's conclusion that "Designers produce designs, methods do not" [91] serves to further emphasize this need.

D. DOCUMENTING DESIGN DECISIONS

In addition to the use of the documentation techniques described previously, there are many other alternatives available. This section will examine four commonly used techniques: HIPO, Flowcharts, Structured Flowcharts and Program Listings. Unlike the first three techniques, the program listings are not actually available until after the design has been implemented. Regardless of this difference, they all are valuable tools in providing the maintenance programmer with a clear understanding of the system and its components.

1. Hierarchy Plus Input-Process-Output (HIPO)

HIPO diagrams were developed by IBM as a design aid and documentation technique. As a result of the stature and size of its originator, it has been widely promoted as an alternative means for documenting design decisions. Described in detail in References [96] and [97], HIPO diagram packages consist of three types of diagrams: a visual table of contents, overview diagrams and detailed diagrams. The visual table of contents gives a graphic display of the major functions to be performed by the system and the relationship between each functional module. The top box identifies the overall purpose of the system. The subsequent levels break that purpose into logical functions and subfunctions. It also contains a legend to define the symbology used in the overview and detailed diagrams. The objective of the overview diagram is to provide general information about the system. It describes the major functions of the system and references the detail diagrams necessary to reduce eacn major function into a series of smaller, single-function modules. The detail diagrams contain the basic elements of the system, describe the specific functions, show the inputs and outputs to each module. In addition to showing data flows, these diagrams also show the flow of control between levels in the hierarchy. Figure 4-12 presents the structure of a typical HIPO package.

Primarily developed as a documentation tool, it is also a useful design aid. It emphasizes the hierarchical and functional aspects of a program and its data flows. It uses a numbering scheme similar to that used in Structured Analysis that can assist maintenance programmers to trace a function from the documentation to the actual code. It also graphically illustrates the interconnection between modules of different levels which is helpful in determining the effect of a change to a given module will have on the rest of

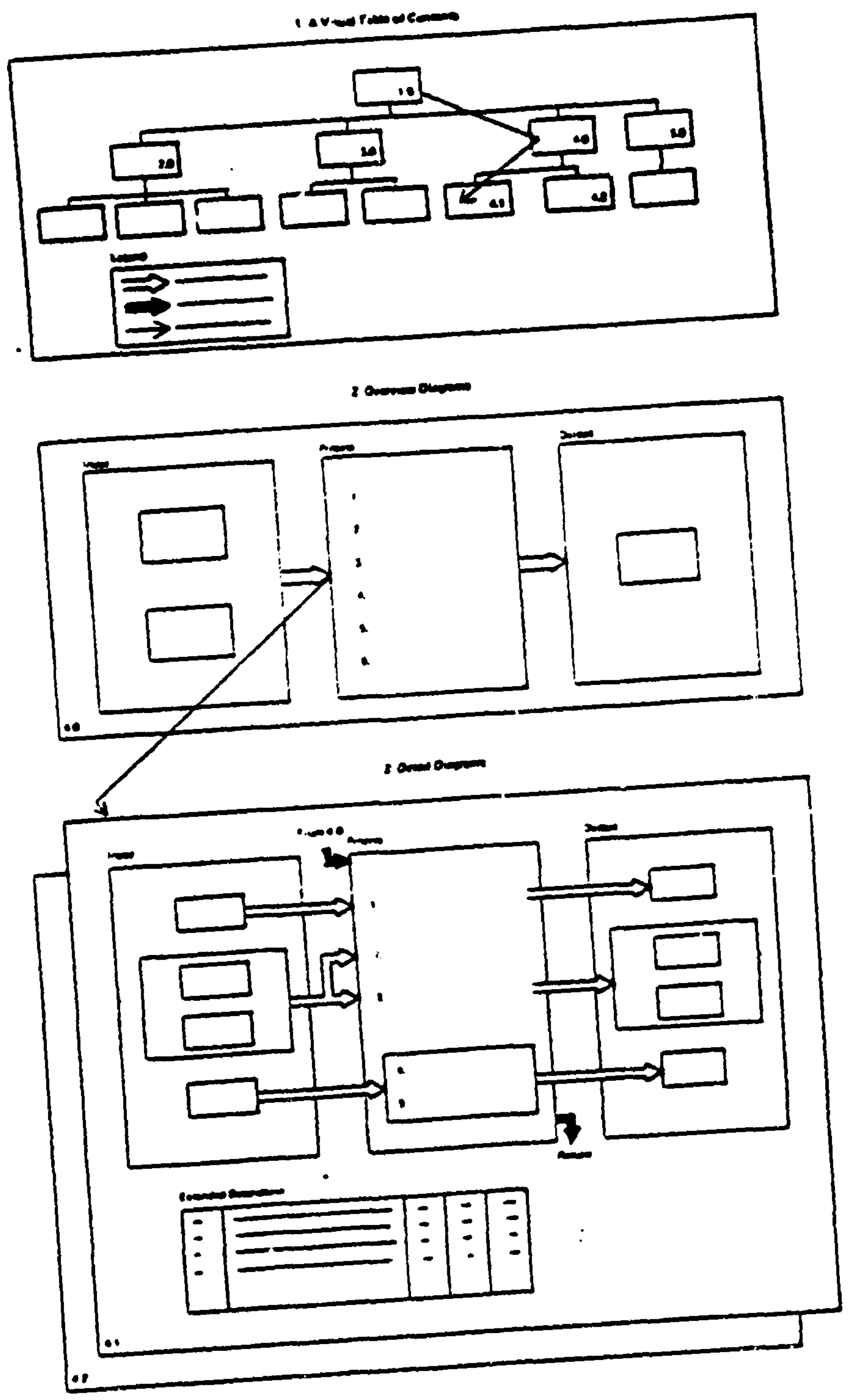


Figure 4-12. A Typical HIPO Package [98].

the system. Another advantage of HIPO diagrams is that the same package can be used repetitively with "...gradual refinements made to the diagrams as additional steps are taken." [34]

2. Flowcharts

Flowcharts are a graphic representation of program logic. Their purpose is to make it easier to see the relationships and flow of control among the various design elements. It is a technique that has been widely used since they were first used by von Neumann for computer applications. The notational symbology has been codified into a set of standards that have been adopted by the federal government and published in FIPS 24 Flowcharting Symbols and Their Usage in Information Processing [98]. Figure 4-13 is an example of a flowchart that describes the process of reading a book and includes loops to show what happens when you are interrupted.

Many authors are opposed to the use of flowcharts. Brooks [52] refers to the practice of using flowcharts as a documentation tool a "curse" and "a most thoroughly oversold piece of documentation." He also labels the practice of requiring designers to deliver flowcharts as an "absolute nuisance." He points out that flowcharts show only one aspect of the structure of a program, the decision structure and that they tend to be difficult to read due to the multiple pages required to document a large program. With regard to the use of flowcharts by maintenance programmers, Weinberg [70] states that "we find no evidence that the original coding plus flow diagrams is any easier to understand than the original coding itself - except to the original programmer." Glass [32] contends that while flowcharting is not particularly useful as a documentation tool, it is a useful visual aid for the designer in representing his thought process. Brooks contends, however, that in actual practice, he has "never seen an experienced programmer who routinely made detailed flowcharts before writing a program...where organizational standards require flowcharts they are almost invariably done after the fact." [52]

DoD guidelines on the use of flowcharts are inconsistent. The Software Acquisition Management Guidebook [71] recommends that DoD not procure flowcharts with software. MIL-STD 1679 states that there is no requirement that flowcharts be deliverable items when contracting for software. In contradiction to these two guidelines, SECNAVINST 3560.1 [78] requires that "a flowchart shall be included for each major procedure or subroutine that depicts operations performed by the subprogram" be included in the Program Description Document. This requirement may be a result of the fact that SECNAVINST 3560.1 was published in 1974, when flowcharts were more in vogue as a documentation tool.

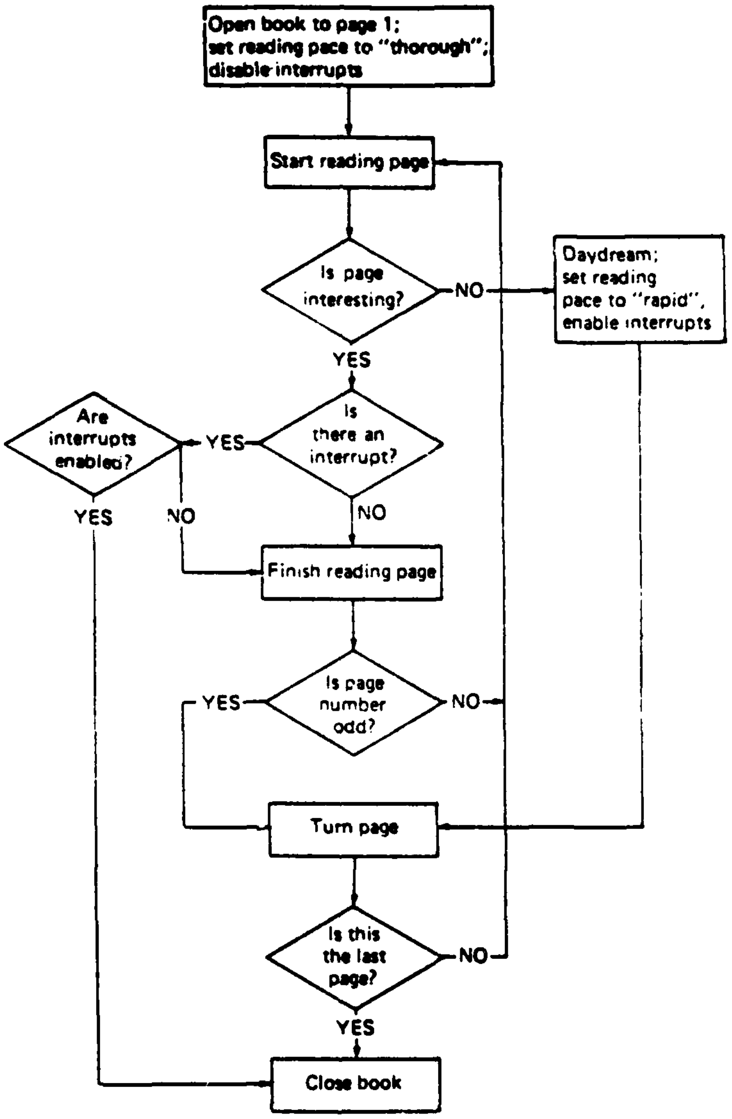
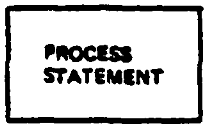


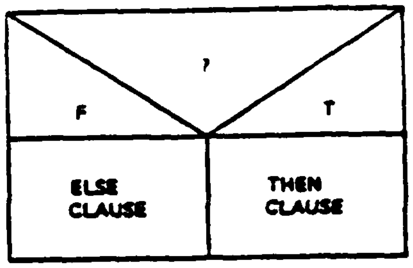
Figure 4-13. Sample Flowchart [32].

3. Structured Flowcharts

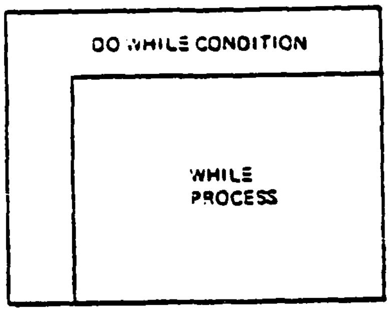
With the advent of structure programming technology and the recognition of the inadequacy of traditional flowcharting techniques, a new form of flowcharts, called structured flowcharts, have been proposed. Alternatively called Nassi-Shneiderman [99], or N-S, flowcharts after their originators, they provide a graphic representation of a program's logic design utilizing the control structures first proposed by Bohm and Jacopini [38]. They can provide a maintenance programmer with a quick reference for finding the code performing a logical function. Yoder and Schrag [100] provide a thorough description of how to utilize this alternative means of documenting design decisions. Figure 4-14 illustrates the basic format of N-S charts which utilize the three basic and two additional program logic control structures mentioned earlier. An example showing how these basic structures can be nested and combined to illustrate a program's processing procedures is provided in Figure 4-15. This figure illustrates how a module that computed FICA deductions in a payroll program would be presented utilizing the N-S format. One great advantage of this tool is that because it uses the same control structures as structured programming, there is a one-to-one correspondence between the N-S flowcharts and the program logic providing structured programming was utilized. This is relevant to DoD since structured programming is required in all federal government software development projects both tactical and non-tactical. It would be of limited



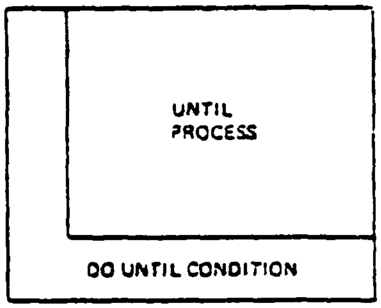
Process Symbol



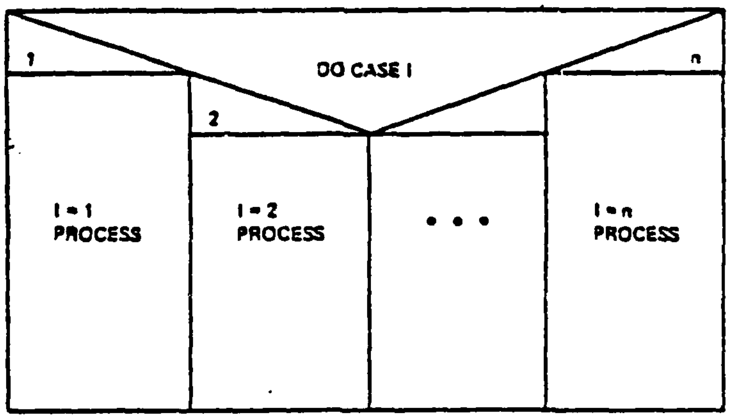
Decision Symbol



DO WHILE Symbol



DO UNTIL Symbol



CASE Symbol

Figure 4-14. Basic N-S Flowchart Format [100].

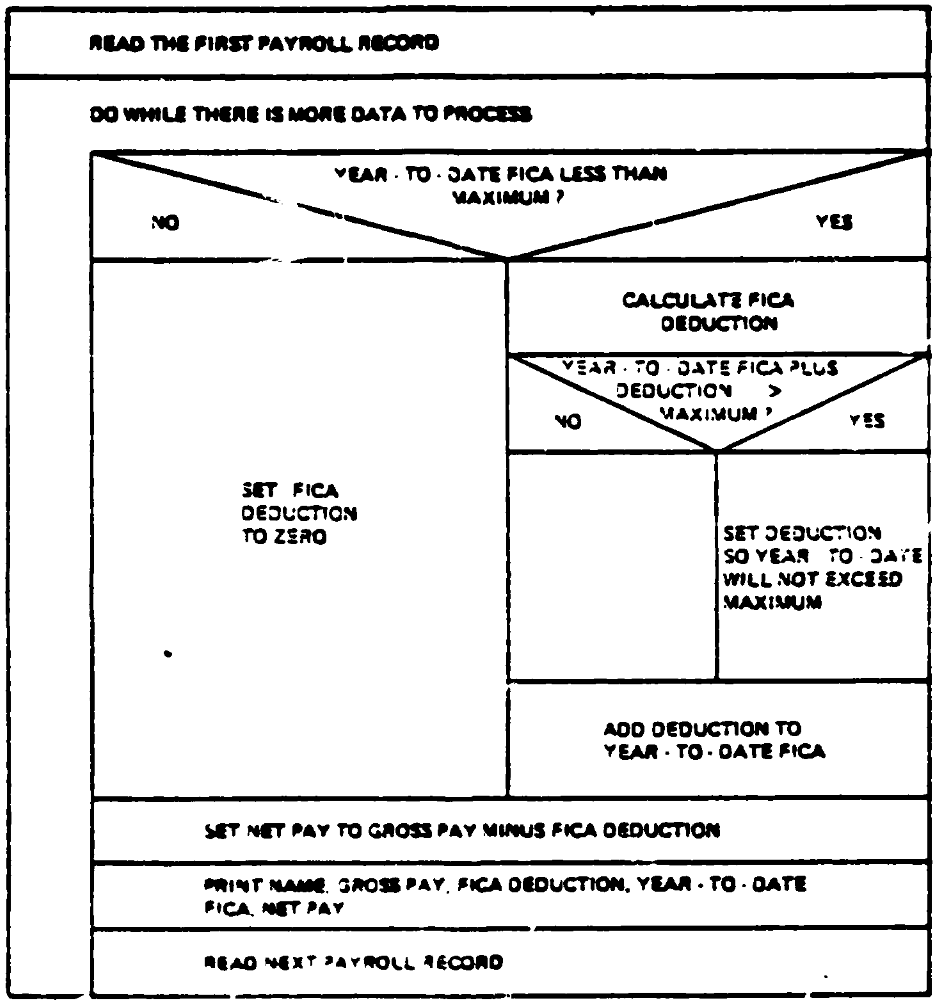

READ THE FIRST PAYROLL RECORD
DO WHILE THERE IS MORE DATA TO PROCESS
YEAR - TO - DATE FICA LESS THAN MAXIMUM ?
NO
YES
CALCULATE FICA DEDUCTION
YEAR - TO - DATE FICA PLUS DEDUCTION > MAXIMUM ?
NO
YES
SET FICA DEDUCTION TO ZERO
SET DEDUCTION SO YEAR - TO - DATE WILL NOT EXCEED MAXIMUM
ADD DEDUCTION TO YEAR - TO - DATE FICA
SET NET PAY TO GROSS PAY MINUS FICA DEDUCTION
PRINT NAME, GROSS PAY, FICA DEDUCTION, YEAR - TO - DATE FICA, NET PAY
READ NEXT PAYROLL RECORD

Figure 4-15. Example N-S Flowchart [100].

utility in those cases where tne program did not use the control structures required such as in programs developed prior to the standard being promulgated. Also since each module is to be contained on a single 8 by 11 form, it serves as added inducement to make the modules small and single-functioned. There is, however, little information in the technical literature to shed light on how well this concept has been accepted or if they required more time to develop or maintain.

4. Program Listings

While program listings are not actually a means of documenting design decisions as they are made, they are included as a means of documentation in that it would be highly desirable if programs could be made self-documenting. This would eliminate the necessity of maintaining multiple forms of documentation that represent the same program logic. Many authors advocate such an approach through the use of structured programming listings, which are computer generated copies of the compiled source code. These listings often include cross-reference listings since they can be automatically generated by the compiler. They are a valuable tool to both the designer and maintenance programmer in that they identify every place an item, such as a variable or subroutine, appears in the programs. These listings can be used to check for extraneous variables that are never called, as well as serve as a reference tool for the maintenance program to determine the interrelationships between parts of the program.

Myers [39], for example, argues that all other forms of documentation are simple redundancies when he states:

"Since we already have the code, why not let it serve as the logic documentation? Additional documentation such as a flowchart would be undesirable because it would be redundant. Redundancy in any type of documentation should be avoided because it increases the chance of conflicts. Furthermore, unless care is taken to update the documentation (which is more difficult if the logic and physically separate from the code), redundant documentation becomes totally useless after the code is modified a few times." [39]

Both MIL-STD 1679 and SECNAVINST 3560.1 contain specific guidelines as to what constitutes a self-documenting program. These documents should be examined in detail to ensure that software delivered meets the requirements set forth in each.

5 Summary

This section has illustrated a variety of documentation tools and techniques that can be used in representing program design. In a survey of documentation techniques conducted in 1979 by Anderson and Shumate [101] to determine which type was found most useful by maintenance programmers, the preferred ranking was, in descending order:

1. Program listings.
2. English language narratives.
3. Flowcharts.
4. Hierarchy diagrams.
5. Data base design documents.
6. HIPO

The trend toward increased emphasis on the use of program listings should continue. Since, however, design decisions must be documented prior to the code ever being generated, it seems unlikely that the need for some graphic means of recording those decisions will be eliminated. If it is true that a picture is worth a thousand words, the use of graphic representations is a way of conveying the designer's intent to the programmer in a concise, yet physchologically reasurring way. Clearly, a wide variety of documentation tools will always be necessary.

E. MANAGEMENT TECHNIQUES

In addition to the tools and techniques available to designers, consideration must be given to the proper mangement of the development process. In the introduction, reference was made to a 1980 GAO study on federal software acquisition projects where nine cases were studied. The single successful project was in no small part due to the presence of a highly capable manager. The software engineering curriculum examined in Chapter II also emphasized that it takes more than knowledge about computer related activities to make a truly capable software engineer. In this section, two management techniques will be examined: the chief programmer team concept developed and promoted by IBM and software configuration management, a means of controlling change to the software throughout its life cycle.

1. <u>Chief Programmer Teams</u>

The software development management technique, known as the "chief programmer team" concept, was developed by Harlan Mills, Terry Baker and their associates at IBM. [102] In an article describing an experiment utilizing this concept [63], two major motivations for trying this approach were cited. One is the realization that because of the rapidly expanding nature of computing, many projects are staffed primarily by inexperienced people. At the same time, those with technical expertise and experience are pushed into higher levels of management where they are able to make only limited contributions to the technical aspect of a project. The second motivation was the observation that little functional specialization is used on a project. In the more traditional approach, a single individual is responsible for designing, programming and testing a single module. The primary feature of the chief programmer team concept is a functional organization centered around a competent, experienced person who has total responsibility for the technical development of the system. The chief programmer, or as Brooks [52] calls him, the "surgeon," personally develops the overall system and programs the most difficult parts.

Other members of the team are chosen and assigned tasks primarily on the basis of whether or not they can extend the capabilities of the chief. A back-up individual is assigned to assist the chief programmer and is experienced enough to take control of the project should, for some reason, the chief programmer become unavailable. Routine jobs such as coding simple programs, removing syntax errors, and running simple tests are carried out by the junior members of the team who have less experience than the chief or his assistant. Clerical duties such as key-punching, running jobs, and maintaining listings are given to a secretary or librarian. In Brook's extended model of the chief programmer team there is one individual assigned to write all of the user documentation in order to retain a consistency throughout the documentation.

In the experiment described by Baker, the size of the group was not large, never exceeding more than 11 people at any time. The experimental group designed and implemented an archival cataloging system for the New York Times that consisted of over 80,000 lines of source code. For larger projects, the division of the total task into separable parts permits the utilization of the functional technique in each of the resultant subtask areas.

As described by Baker, there are three additional components of the chief programmer team: the use of automated program support libraries, top-down programming and structured programming. The use of a programming support library is intended to isolate purely clerical work from the technical aspects of system development. The programming support library consists of four major parts. There is an "internal" library of source code, load modules, and test cases in machine-processable form. An "external" library contains listings of the internal library and records of superceded versions of the system. These expired versions provide a record of past decisions and can be a very valuable tool in avoiding making the same mistakes over again. A set of "machine procedures" for updating libraries, retrieving modules, testing and so on is the third major piece of the programming support library. Finally, there is a set of "office procedures" that are followed by the clerical personnel to aid in maintaining and adding to both the internal and external libraries.

The top-down design and structured programming refer to the manner in which the software was designed and implemented. The major functions of the system were identified and expressed in terms of lower level primitives. This process continues until all of the functions are identified in terms of such a sufficiently low level that their implementation can be expressed in a minimal number of programming statements. The structured design aspect of this project refers to Dijkstra's concept of writing programs as a nested set of single-entry, single-exit modules using logical constructs limited to those discussed earlier. Within each of the modules, the nested constructs use a style of indentation like that of Structured English to enhance the readability and understandability of the program for both the other designers and maintenance programmers.

As noted by Baker, the chief programmer team basically contains nothing new. Its contribution, in his view, is that it has integrated, for the first time, four existing techniques: functional specification, program support libraries, top-down design and structured progamming [103].

Advocates of this technique, such as Brooks, claim that it is an effective technique that ensures the conceptual integrity of the design by having one individual responsible for designing the entire system. Furthermore, by reducing the size of the teams, communication cost, in terms of time and interoffice volume, are reduced. The use of structured programming technique and high level languages also increase the efficiency of the programmers and maintenance personnel. One possible disadvantage to this approach lies in the fact that the chief programmer must be a technically superior individual. If he becomes disenchanted with the project or is lured away by higher salary, it could leave his team in a bind, depending on how capable the assistant really was.

2. Software Configuration Management

"Configuration management is," according to Glass [32], "an established recognized engineering discipline in industry, having been applied to the whole range of hardware projects for years." It is a discipline that identifies, baselines, controls and reports changes to what are termed

Configuration items. The need for a managerial tool for controlling the changes to software has, unlike hardware, only been recently recognized, as maintenance costs have risen to their current level.

Bersoff and others [103] have identified four basic elements of software configuration management: configuration identification, configuration control, configuration status accounting, and configuration auditing.

Configuration identification consists of recognizing and labeling the configuration items at selected times, or baseline, during the software lifecycle. While recognizing that change is inevitable, baselining means to "freeze" the software requirements and documentation at predetermined times and using those as standards by which changes can be measured. This need to freeze the solution at certain points have led many to recognize that software development actually leads to a prototype or trial system. The key to having a prototype that requires minimal rework is to fully analyze the user's requirements to make them as complete as possible. However, as Lientz and Swanson [41] point out, regardless of how complete the analysis and specifications are, there is still a great deal of change that evolves from changes in the user's long-term goals and needs, as well as increasing user familiarity with the system leading to enhancement requests. They contend that rather than concern themselves with finding ways to perfect methods of specifying requirements, software designers should concern themselves more with building systems that can be changed readily since the change is inevitable anyway. While agreeing with the idea that the maintainability is a crucial issue, it is the contention of this thesis that the better the specification and analysis, the less corrective maintenance will be required to overcome shortcomings in the system. This reduction allows the maintenance programmer to concentrate on enhancing the software that results from users' requests.

Configuration control provides the means to manage changes to the configuration items and consists of three basic ingredients:

"1. Documentation such as administrative forms and supporting technical and administrative material for formally precipitating and defining a proposed change to a software system.

2. An organization for formally evaluating and approving or disapproving a proposed change to a software system.

3. Procedures for controlling changes to a software system." [103]

Configuration status accounting provides the mechanism for maintaining a record of how the software has evolved as well as giving the status of the system at any stage of implementation. Configuration auditing provides a means to determine how accurately the software product matches its associated documentation. It is a management tool for determining where further documentation maintenance is required.

DoD has recognized the value of software configuration management as a means of controlling changes to the software as well as assuring the quality of the product. DoD Directive 5000.29 Management of Computer Resources in Major Defense Systems, states in part:

> "Defense system computer resources, including both computer hardware and software will be specified and treated as configuration items." [104]

Additionally, MIL-STD 1679, even more explicitly, requires contractors to:

> "...establish and implement the disciplines of configuration management; namely configuration identification, configuration control, and configuration status accounting. The contractor shall be cognizant of the requirement for long-term life-cycle support of the weapon system software." [105]

MIL-STD 52779 (AD) [106] further requires that the contractor provide for independent audits to ensure that the objectives of the configuration control program are attained.

As Bersoff notes, the problem with software configuration management is that it has generally fallen under the "umbrella" of the configuration management of the entire system. Hardware, being more visible and tangible, has been treated in great detail. Software, on the other hand, being less visible and tangible, has been largely neglected. Fortunately, software configuration management has been accepted as a vital tool in managing change. In fact, as stated earlier, Kline [27] has recommended replacing the phrase "software maintenance" with "software configuration management" to emphasize the crucial role it plays in the maintenance of software. With regard to controlling change, Lindhorst [107] has proposed "scheduling" maintenance of software systems. Scheduled maintenance is a policy where instead of judging each request for change as it is received, all requests for changes to a particular application is held until a predetermined time. At this predetermined time, all of the proposals are evaluated both on their individual merit and with respect to each other. Lindhorst has installed this concept as part of the configuration management program at a midwestern bank. Among the benefits he perceives from this scheduling technique are:

1. The consolidation of requests so that all source code and documentation for a particular application can be updated at one time.

2. It forces the user departments to think more about the changes they are requesting.

3. It provides the controlling organization with the opportunity to evaluate all proposed changes at one time, giving them more control over the system.

While giving more control and consolidating the evaluation process, there may be a disadvantage in that if the schedule is enforced too rigidly, problems that require immediate attention may be postponed too long, thus increasing user frustration with the system.

F. SUMMARY

As has been illustrated throughout this section, there exists a wide variety of technical and managerial tools and techniques that are available to the project manager. Some, such as configuration management are fr quently accepted as a valuable tool in any software project. Others, such as the use of structured flowcharts or design methodologies that are predicated on the structure of program logic and data, are less accepted and require further research in order to validate their ability to produce software that is both reliable and maintainable. Since some of these tools are more appropriate for projects of considerable scale, e.g. the automated specification and design methods such as SREM or PSL/PSA, the project manager must weight the benefits from using any of these tools against the risks involved and perhaps tailor them to better meet the needs of the sponsoring organization.

V. DOCUMENTATION/MAINTAINABILITY

A. INTRODUCTION

The purpose of this chapter is to examine the effect of documentation on the maintainability of a software product. Documentation, properly designed, can be used by a project manager as a means of determining whether or not the product will be easily adapted or modified and if the user's original requirements are being met.

Another purpose of this chapter is to examine two proposed methods to measure maintainability. This chapter will examine the possibility of extending these approaches so that they can be used as a means of determining the maintainability of a software product earlier in the life cycle. The reason for this extension is that, as stated earlier, this quality characteristic must be considered from the beginning of a software development project. Thus, by having a means of measuring the maintainability as early as possible, it both emphasizes the need for early consideration and as such can provide a means for the project manager to ensure that the delivered product possesses this vital characteristic.

Applicable instructions and standards that exist within the Department of Defense which affect the documentation requirements will be presented. Federal Information Processing Standards (FIPS) Publication 38 and Military Standard-1679 will be discussed, with concern to what changes should be made to them, so that they may better ensure that the documentation developed throughout the software life cycle aids in achieving maintainability.

B. DOCUMENTATION

The importance of documentation has not always been acknowledged. This has resulted in programs where requirements documentation, or specifications did not exist, or were out of date. The three main reasons why this inadequacy has occurred are attitude, time and method.

As Vaughn [13] points out there has been a widespread belief that documentation for a complex system could be put together quickly after the system was tested. It was found out that this is not the case. It was pointed out that time was the reason why most documentation efforts foundered, and that 20 percent of total development time should be allowed for documentation.

Brooks [52] on the other hand felt that the lack of documentation occurred because of the methods used to document software were inadequate, resulting in programmers not knowing how to develop good documentation.

The attitude towards documentation has changed, as its importance in program devleopment became recognized. One reason for this change was the realization that documentation contained information which was necessary to be provided to maintenance programmers in order for them to make changes to the existing software in a more efficient manner. As the

Defense Logistic Agency, technical report, Software Acquisition Management Guidebook: Software Maintenance, points out, documentation "provides the necessary technical and status information." [71]

Good documentation, however, can be used for much more than providing maintenance programmers with the necessary technical information needed to effect changes. During the development phase documentation can also be used as a communication tool, among designers, users and programmers. It is a means for preventing duplication of effort and it can be used as a basis for design reviews.

During the maintenance phase it can be used to evaluate the feasibility of changes, as a guide to find and correct errors, as a repository of design information and as a means of preserving the program's conceptual integrity.

Good documentation possesses the characteristics of being easy to use, understandable, [71] modifiable and traceable, [108] as was discussed in earlier chapters. In order to obtain these characteristics the documentation must be properly designed to ensure that it adequately addresses the audience for which it is intended.

1. Designing Documentation

There is a broad range of opinions on how documentation should be designed. They very from the statement that one should "document unto others as you would have them document unto you" [118] to proposals by Peters [110] for a software design documentation standard. Others such as

Glass [32] believes that documentation should meet the guidelines and standards imposed by the United States Government. His contention is that since the federal government has been most thorough on what software documentation should contain and that military projects normally "include documentation requirements on the same level of importance as the requirements for the software product itself", these standards represent the best approach to providing the needed documentation. Even so, Peters [110] states that many of the schemes for design documentation in use today fail to address the nature of the problem. "They put forward approaches which will solve the software design documentation problems without relating what the characteristics are which caused this standard to take this form."

In an attempt to specify information requirements that must be met by a software documentation design, it was proposed, in Reference [60], that software design documentation should be developed from a consistent set of design principles. For this reason Heninger [60] suggests that the first step in designing documentation to be useful "explicit decisions must be made about the purposes it should serve. Decisions about the following questions affect its scope, organization and style: a) What kinds of questions should it answer? b) Who are the readers? c) How will it be used?

Heninger has proposed the following three design principles in the development of documentation.

1. "State questions before trying to answer them." This should be done at every stage of writing the requirements. She states that "if this is not done, the available material predjudices the requirements investigation so that only the easily answered questions are asked."

2. "Separate concerns." This principle serves "the objective of making the document easy to change, since it causes changes to be well-confined. For example, hardware interfaces are described without making any assumptions about the purpose of the program; the hardware section would remain unchanged if the behavior of the program changed."

3. "Be as formal as possible." This was in order to present information as precise, concise, consistent and complete as possible. Also, by using a formal notational scheme, the document will serve as a basis for formal proofs of correctness.

Using these design principles and the answers to the above questions the objectives of a document can be adequately developed. By developing the objectives, the design of the documentation can be evaluated based on how well they meet those objectives. As a demonstration, she specifically derived six objectives for the requirements document to be used on the A-7 flight program:

1. Specify external behaviour only.
2. Specify constraints in the implementation.
3. Be easy to change.
4. Serve as a reference tool.
5. Record forethought about the lifecycle of the system.
6. Characterize acceptable responses to undersired events.

While the objectives of the requirement documentation have, thus, been stated, Schneidewind points out that another problem is that the requirements objectives should include precise performance goals. As he states in Reference [67] "What is needed is a clear statement of performance goals in the user requirements statement, consistency in the use of these goals in subsequent stages of development and the ability to trace these goals forward from user requirements phase to maintenance phase; and backward, from maintenance to user requirements."

2. Documentation Requirements

Brooks [52] states that "different levels of documentation are required for the casual user of a program, for the user who must depend upon a program, and for the user who must adapt a program from changes in circumstances or purpose." It is then necessary to design each individual document in order ensure its understandability by the intended reader. For example, while the program design documentation could be written given the assumption that it would be used by individuals familiar with computer programming, the user's manual would have to be written for an assumed less skilled audience. Glass [32] states that software documents required for the government falls into four broad categories: planning documents, administrative procedures, software test procedure/ reports and support documentation. Young [109] further subdivides these levels of documentation into the types which are listed below:

1. Functional Requirements Document.
2. Data Requirements Document
3. System and Sub-System Specifications.
4. Program Specification.

5. Data Base Specification.

6. Test Plan.

7. User Manual.

8. Operations Manual.

9. Program Maintenance Manual.

10. Test Analysis Report.

Young's subdivision concides with the FIPS Pub 38 documentation. The following definitions and statement of purpose for each document type, as described in Reference [111], is given below:

Functional Requirements Document. "The purpose of the functional requirements document is to provide a basis for the mutual understanding between users and designers of the initial definition of the software, including the requirements, operating environment, and development plan."

Data Requirements Document. "The purpose of the data requirements document is to provide, during the definition state of software develcpment, a data description and technical information about data collection requirements."

System/Sub-System Specification. "The purpose of the system/sub-system specification is to specify for analysts and programmers the requirements, operating environment, design characteristics, and program specifications for a system or sub-system."

Program Specification. "The purpose of the program specification is to specify for programmers the requirements, operating environment, and design characteristics of a computer program."

Data Base Specification. "The purpose of the data base specification is to specify the identification, logical characteristics, and physical characteristics of a particular data base."

User Manual. "The purpose of the user manual is to sufficiently describe the functions performed by the software in non-ADP terminology, such that the user organization can determine its applicability and when and how to use it. It should serve as a reference document for preparation of input data and parameters and for interpretation of results."

Operations Manual. "The purpose of the operations manual is to provide computer operation personnel with a description of the software and of the operational environment so that the software can be run."

Program Maintenance Manual. "The purpose of the program maintenance manual is to provide the maintenance programmer with information necessary to understand the programs, their operating environment and their maintenance procedures."

Test Plan. "The purpose of the test plan is to provide a plan for the testing of software; detailed specifications, descriptions, and procedures for all tests and test data reduction and evaluation criteria."

Test Analysis Report. "The purpose of the test analysis report is to document the test analysis results and findings, present the demonstrated capabilities and deficiencies for review, and provide a basis for preparing a statement of software readiness for implementation."

This list although well conceived is not all encompassing as advancements in software engineering are made. For this reason Schneidewind [67] proposes that an "Interfaces Specification" document should be added to the list.

In support of the life cycle models that have been constructed for design reviews, additional documentation requirements have been suggested. A brief discussion of a few type of documents that might be included are given below as defined by Glass [32].

Computer Program Development Plan. It provides a top-level overview of the organizational responsibilities, project phasing, and major tasks to be accomplished during the software development process.

Facilities Management Plan. It will include specifications for computers, peripheral equipment, office and laboratory space.

Configuration Management Plan. It specifies three planning activities associated with reliability.

(1) Software release levels defined by degrees of testedness.

(2) Problem reporting and retesting required after a change is made to a software test article.

(3) Responsibilities for test reporting, and test monitoring.

3. Documentation/Maintainability Instructions

Within DoD a number of formal standards, and instructions have been developed concerning the production of documentation for use in the maintenance of software. These various DoD requirements vary in applicability between software developed for tactical systems and software developed for ADP systems. A number of the more predominate documents from each community will be presented.

a. Tactical Maintenance Documentation Standards

(1) SECNAVINST 3560.1 Combat System Program Description Document. This instruction on documentation, written in 1974, is specifically designed for weapon system software. It consists of three documents which support the maintenance of tactical software. These documents are called the (PP) Program Package, (DBD) Data Base Design and the (PDD) Program Description Document.

(2) MIL-STD-52779 (A.D.) Software Quality Assurance Program Requirement. This standard, developed in

1974, requires a quality assurance plan to be implemented specifically for the development of programs and related documentation. Although concerned primarily with the development phase, it is also important to software maintenance in that it directly affects the quality of the initially delivered product.

(3) <u>MIL-STD-483 (USAF) Configuration Management Practice for System and Equipment</u>. This standard defines the activities associated with controlling changes to computer programs.

(4) <u>DODD 5000.29 Management of Computer Resources in Major Defense Systems</u>. This directive, issued in 1976, establishes policy for the management of computer resources during system acquisition. Consideration of the maintainability of software is singled out as a primary consideration during the initial design. It also requires that maintenance support items be specified as deliverables in an acquisition project.

(5) <u>MIL-STD-1521 (USAF) Technical Review and Audit for System Equipment and Computer Programs</u>. This standard concerns reviews and audits and how they can be used as a basis for checking compliance with maintainability requirements. It delineates the requirements for the conduct of technical reviews and audits in conjunction with the documents defined in MIL-STD-483.

(6) MIL-STD-1679 Weapon System Software Development. In contrast with other standards issued by various DoD agencies, the issuance of MIL-STD-1679, in 1978, has been widely acclaimed as one of the few, such documents that reflect state-of-the-art concepts. Providing uniform standards for developing weapons system software within DoD, it requires such items as the use of structured programming techniques, configuration management and limits the use of patching in tactical software.

However, because it is aimed primarily at the initial specification design and testing phases, it does not specifically address the consideration of maintainability. Rather, it emphasizes tools and techniques that will optimize the initial development. As Schneidewind [67] points out, this optimization of development is often done to the detriment of maintainability. It should be revised to specifically ensure that maintainability is considered as a primary goal throughout the development.

b. ADP Maintenance Documentation Standards

(1) DoD STANDARD 7935.1-S Automated Data Systems Documentation Standard. This instruction is the basis for systems level documentation of an automated information system. It provides guidelines for the development of a program maintenance manual and is to provide the maintenance programmer with the information necessary to effectively maintain a system. The orientation of this document is mostly towards data base systems.

(2) <u>SECNAVINST 5233.1B Department of the Navy Automated Data Systems Documentation Standard</u>. This instruction promulgates a documentation preparation standard to the Navy based upon DoD INSTRUCTION 7935.1-S previously mentioned.

(3) <u>NMPC-16 INSTRUCTION 5231.1 NMPC-16 Life Cycle Management for Automated Information System Development and Enhancement</u>. Life Cycle Management is the process of administering and an Automated Information System over its entire life with emphasis on strengthening the early decisions that shape automated information systems costs and utility.

The LCM process involves five phases separated by decision milestones and specifies the planning and management functions that must be satisfied and documented to reach each milestone.

Proposed automated information systems are divided into levels according to estimated costs. The guidelines and documentation requirements become more extensive at each higher lever (i.e., as the estimated costs increase).

(4) <u>Federal Information Processing Publication 38</u>. FIPS PUB 38, issued in 1976, by the National Bureau of Standards, is the primary documentation standard for all non-tactical software within the federal government. FIPS PUB 38 was originally published as a guideline implying considerable flexibility in its use. It is not intended as a tutorial or as a guide for clear and concise technical writing. The content of the document types described is sufficiently

general to be applicable to a wide audience in the federal government and for use in a variety of projects. One difficulty that has evolved from this attempt at generality is that as a result, it does not provide sufficient guidance to a specific project. Recognizing this difficulty, a workshop was organized in the summer of 1980 to review the experience in the use of FIPS PUB 38 as part of a more general, five-year review cycle of all the various FIPS publications. Sponsored by the Federal ADP User Group and special interest group on ADP Standards and Quality Assurance, this conference was intended to identify specific criticisms and suggestions in order to provide data for future revisions of this standard. Some of the revisions suggested by participants in this workshop are:

1. Include example of graphic methods to describe functional requirements.
2. Include user acceptance criteria in the test report.
3. Include the expectation of changes to requirements.
4. Relate documents to system life cycle management activities.
5. Reorganize the content guidelines to provide for traceable and consistent presentation of requirements.
6. Begin preparing the user manual during the design state.

Additionally a recommended general approach was developed to solve the problems reported with FIPS PUB 38. The approach consists of three main thrusts: (1) considering documentation althogether and in its wider context versus as individual documents, (2) facilitating a variety of approaches to the development, and (3) modernizing the content.

4. General Comments

These software standards have improved the development and design of software, but generally fail to emphasis the need to achieve maintainability. While each standard or instruction will not be systematically dissected for its faults, the defects of the group as a whole will be discussed.

Many of today's standards and instructions were promulgated when their purpose was for emphasizing critical aspects within program development and not maintainability. Since the pitfalls of not designing for maintainability were not known at the time, these standards and instructions have not ensured maintainability is designed into documentation.

Although attempts have been made through quality assurance plans for improved documentation by stating that the (SDD) Software Design Description shall describe the major components of the software design, it does not include such important aspects as the design decisions themselves or formal specifications of which was expressed in earlier chapters as a must if a project manager wants to be assured of a verifiable end product.

Other engineering fields have made it mandatory for numerous reviews of projects for technical feasibility, such as the (PDR) Preliminary Design Review which is held to evaluate the technical adequacy of the preliminary design of the software, but there is not a (PDROD) Preliminary Design Review of Documentation to evaluate it for traceability, modifiability, understandability or conciseness.

Although FIPS PUB 38, as an example, includes Young's list of documentation applicable to computer software it fails to require the idea of specifically documenting interface specifications, the importantance of which was discussed earlier.

Although attempts have been made to determine documentation requirements prior to or concurrent with designing of the entire system as in the Functional Requirements Document and the Data Requirements Document there is no congruency between the two. The Data Requirements Document does not physically map into the Functional Requirements Document. A result of this lack of physical mapping is that it may be difficult to trace back from the actual source code listing to the appropriate section of the accompanying documentation. Figures 5-1 and 5-2.

In conclusion, it was found that documentation can be used as a means for measuring maintainability. However, to achieve this goal the documentation must be designed to include certain characteristics, and must adhere to the software objectives over its life cycle. The documentation that is required varies, but there are generally acceptable norms that are practiced.

Figure 5-1. Data Requirements Document [111].

Figure 5-2. Functional Requirements Document [111].

Figure 5-2. Functional Requirements Document [111] (Cont'd).

C. AIR FORCE'S EVALUATIONS HANDBOOK

The United States Air Force has developed a methodology for studying software and rating it as to its maintainability. The method is described in the Software Maintainability Evaluator's Handbook which was prepared by the Computer/ Support Systems Division at the U.S. Air Force Test and Evaluation Center, Kirtland Air Force Base, New Mexico.

The purpose of the handbook as stated in Reference [112] "is to provide to the software evaluator the information needed to participate in the Air Force Test and Evaluation Center's (AFTEC's) software maintainability evaluation process." The handbook states that "software maintainability is determined by those characteristics of software and computer support resources which affect the ability of software programmer/analyst to change software." It states that such changes are made to:

a. Correct errors.

b. Add system capabilities.

c. Delete features from programs.

d. Modify software to be compatible with hardware changes.

The handbook is divided into three parts. The first part provides the evaluator with: (1) a background of the AFTEC software maintainability evaluation concept, (2) a basic understanding of the evaluation procedures, and (3) detailed instructions for using AFTEC's standard software maintainability questionnaires and answer sheets. The second part contains the questionnaires and explanatory information on each question. This information is provided in an attempt to ensure the evaluator fully understands the intent of each question. Included are definitions of terms, examples, explanations, and special case resonse instructions, as necessary. The third part of the handbook is a cross reference index.

The Air Force states that "the methodology for evaluating software maintainability is based on the use of closed form questionnaires with optional written comments. These questionnaires are designed to determine the presence or absence of certain desirable attributes in a given software product." [112] The elements of software maintainability and their relationships as used by the Air Force are shown in Figure 5-3 and are described in the following paragraphs. Figure 5-4 is a sample questionnaire used in evaluating documentation for modularity. A complete listing of all of the questions used in this method is contained in Appendix A. The Air Force contends that the hierarchical evaluation structure allows them to identify potential maintainability problems at various levels. These levels include three software categories (documentation, source listings, support resources) as well as six quality attributes that they claim directly affect maintainability.

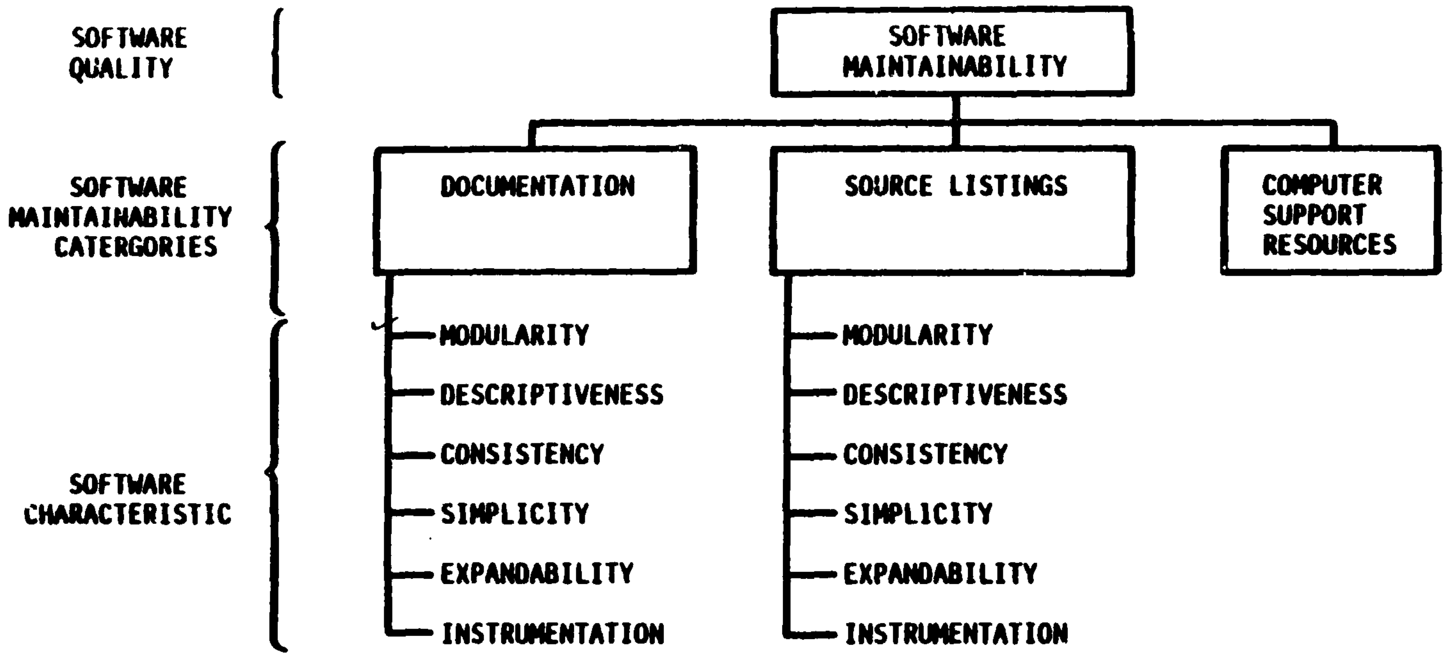


Figure 5-3. Elements of Software Maintainability [112].

QUESTION DATA SHEET

Question Number D-12

QUESTION: The documentation indicates that program error processing is done by one set of modules designed exclusively for that purpose.

CHARACTERISTIC: Modularity (processing modularity).

EXPLANATIONS:
The documentation describing the program functions and control flow should also describe how the program processes any error condition (e.g., via execution of one error processing module or not). Checks of module processing may indicate whether any error processing functions are mixed with other application functions.
It is best if each function, module, submodule, etc. does not handle its own error processing unless adequate corrective measures are appropriate. There should be one part (e.g., a few modules) of the program which is for error processing.

EXAMPLES: Editing of input data should be documented.

GLOSSARY: Error processing: The steps required to set program data and control states following the detection of an error condition.

SPECIAL RESPONSE INSTRUCTIONS: If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for error processing, then appropriate agreement with the question should be so indicated. If in addition, these error processing modules are systematically organized as an error processing function, then there should be essentially complete agreement with this question statement.

Figure 5-4. Question Data Sheet [112].

1. Software Categories

The Air Force defines software as consisting "of a set of computer instructions and data structured into programs, and the associated documentation on the design, implementation, test, support, and operation of those programs." Each program is separately evaluated. For each program there are related categories which are evaluated for the characteristics which affect its maintainability. The categories are the software documentation, the software source listings, and the computer support resources. The Air Force emphasizes that only "products that will be available to the maintenance programmer are to be considered in an evaluation."

a. Software Documentation

The Air Force defines software program documentation as "the set of requirements, design specifications, guidelines, operational procedures, test information, problem reports, etc. which in total form is the written description of a computer program." The primary documentation which the Air Force uses in this evaluation consists of the documents containing program design specifications, program testing information and procedures, and program maintenance information. The documents are evaluated both for content and for general physical structure. The content evaluation is primarily concerned with how well the overall program has been designed for maintainability. The format evaluation is primarily aimed at how the physical structure of the documentation aids in understanding or locating program information.

b. Software Source Listings

The Air Force defines software source listings as "the computer generated form of the program code in its source language (e.g., Fortran, Cobol, Jovial, Ada, assembly language, etc.)." Since the source listing represents the program as implemented, in contrast to the documentation which for the most part represents the program design or implementation plan.

c. Computer Support Resources

The Air Force defines computer support resources to "include all the relevant resources such as software, computer equipment, facilities etc., which will be used to support the maintenance of the software being evaluated." The characteristics of and procedures for the evaluation of computer support resources, however, are contained in a separate document.

2. <u>Software Maintainability Test Factors</u>

The Air Force determines the maintainability of software documentation by examining six characteristics or test factors: modularity, descriptiveness, consistency, simplicity, expandability, and instrumentation. The following definitions, used by the Air Force to provide guidance to the evaluation teams, are as follows. A discussion of these applications in evaluating their documentation as well as differences between the Air Force definitions and those provided earlier is also provided.

a. Modularity

"Software possesses the characteristic of modularity to the extent that a logical partitioning of software into parts, components, and/or modules has occurred."

The Air Force uses this characteristic because it states "software that is the easiest to understand and change is composed of independent modules." Using this reasoning, each software product is therefore evaluated in relation to the extent to which its logical parts, components, and modules are independent. It states that "the fewer and simpler the connections between parts, the earier it is to understand each module without reference to other parts. Minimizing connections between parts also minimizes the paths along which changes and errors can propagate into other parts of the system."

b. Descriptiveness

"Software possesses the characteristic of descriptiveness to the extent that it contains information regarding its objectives, assumptions, inputs, processing outputs, components revision status, etc."

The Air Force believes this characteristic is important in understanding software. It states that "documentation should have a descriptive format and contain useful explanations of the software program design."

c. Consistency

"Software possesses the characteristic of consistency to the extent the software products correlate and contain uniform notation, terminology and symbology." This emphasis on uniform notation is consistent with the definition of uniformity provided by Ross [23] in Chapter III.

The Air Force states that "attention to consistency characteristics can greatly aid in understanding the program." As an example, the Air Force states "programs using consistent conventions require that the format of modules be similar. Thus by learning the format of one module (preface block, declaration format, error checks, ect.) the format of all modules is learned."

d. Simplicity

"Software possesses the characteristic of simplicity to the extent that it lacks complexity in organization, language, and implementation techniques and reflects the use of singularity concepts and fundamental structures."

The Air Force states "the aspects of software complexity (or lack of simplicity) that are emphasized in the evaluation relate primarily to the concepts of size and primitives. The less there is to discriminate and the more use there is of basic or primitive techniques, structures, etc. the simpler the software will tend to be."

e. Expandability

"Software possesses the characteristic of expandibility to the extent that a physical change to information computational functions, data storage or execution time can be easily accomplished once the nature of what is to be changed is understood." This is consistent with the definition of evolvability that was provided in Chapter III.

The Air Force uses this characteristic because it states "software may be perfectly understandable but not easily expandable. If the design of the program has not allowed for a flexible timing scheme or a reasonable storage margin, then even minor changes may be extremely difficult to implement."

f. Instrumentation

"Software possesses the characteristic of instrumentation to the extent it contains aids which enhance testing."

This characteristic is used because from the Air Force viewpoint "this part of the evaluation reflects the concern that the software be designed and implemented so that instrumentation is either imbedded within the program, can be easily inserted into the program, or is available through a support software system, or is available through a combination of these capabilities."

3. The Air Force Measurement Technique

The Air Force's measurement technique rates the various maintainability considerations of software on a multipoint scale. This procedure was developed because in the past the approach the Air Force used to evaluate software documentation had not been qualified statistically. The previous method consisted of a single knowledgeable person who examined the documentation and provided an interviewer with a subjective appraisal. This interviewer in turn, would make his own subjective interpretation of the evaluator's remarks.

The present approach rates the questions presented earlier on a six-point response scale where six is the highest possible score and one is the lowest. The questions have been grouped into several test factors, and the scores for all questions applicable to a given test factor were averaged to obtain the score for that factor. Each factor is assigned a relative weight, based on its importance, to arrive at an overall score. Thus, the measures of effectiveness scores are straight averages for test factors and the weighted averages of these test factor scores for documentation. The thresholds were determined to be 3.3 and the goal 5.0 on the six-point scale. Figure 5-5 is an example of the results of a software maintainability assessment.

Software Maintainability Assessment

Item Rated	Score	Evaluation Criteria Threshold	Evaluation Criteria Goal
Maintainability	3.84	3.30	5.00
Documentation	3.27*		
Modularity	2.64*		
Descriptiveness	3.25		
Consistency	3.88		
Simplicity	4.48		
Expandability	3.64		
Instrumentation	1.93*		
Source Listings	4.15		
Modularity	5.04		
Descriptiveness	3.83		
Consistency	4.06		
Simplicity	4.47		
Expandability	4.41		
Instrumentation	2.59*		
Computer Support Resources	3.60	2.80	4.50
Support Software	3.40	3.30	4.70
Support Equipment	3.30	3.00	3.70
Building	4.20	2.00	5.00

*Below Threshold

Figure 5-5. A Software Maintainability Assessment [113].

4. Critique of the Air Force's Methodology

One problem with the Air Force evaluation methodology is that it is not used until the latter portion of the accept- and testing phase of the acquisition process.

The Air Force's methodology of waiting until the design has been coded will cause the cost of making any changes to rise.

The procedure by which the Air Force measures maintainability remains very subjective in nature in that each evaluator assigns the points of the grading criteria on his or her own judgement. Since in the early phases of software development there is no visible output except documentation, major emphasis must be made in evaluating each document as it is developed. The best way of determining if documentation is of sufficient quality to reduce design errors is to determine if it was designed, by the means presented earlier and by answering the questions developed by Heninger.

Another fault with the Air Force's approach is that it fails to determine if the performance goals have been stated in the user requirement document. It therefore cannot be determined if this goal is consistent throughout the subsequent stages of the software's development. Without looking at documentation early it cannot be determined if decisions affecting the scope, organization and style of the documentation have been made that meets the objectives of the requirements document. This is not possible however, since the objectives of the user requirements have not been determined.

The Air Force makes no attempt to determine if design decisions have been recorded. Parnas [30] however, emphasizes that since the order in which design decisions are made effects the structure and maintainability of the software, because information resulting from early design decisions is used in making later decisions. A precise record of the intermediate design decisions is essential. The reason it is essential can be explained by Daly [10] who states that "the development cost required to detect and resolve a sotfware bug after it has been placed into service is thirty times larger than the cost required to detect and resolve a bug..." in the design phase.

He states the following reasons by bugs cost more to correct after a program has been released to the customer, as in the case of the Air Force, than during the early phases.

1. After commercial release- problems are usually more complex.

2. After commercial release- problems are reported as system malfunction; an effort must be spent to translate problems into a software bug.

3. After commercial release- many problems are resolved by design maintenance programmers rather than the original designer. Design maintenance programmers must spend effort reviewing detailed code.

4. After Commercial release- problems require more definition and more formal documentation. Formal test plans and multilevel testing must be performed to ensure that accurate corrections reach the field.

5. After commercial release- problems resolution must share the heavy overhead cost for configuration management.

The Air Force has attempted to ensure that modern software engineering principles are used.

Although the Air Force's definition of these characteristics are generally accepted, there are other aspects that should be considered. For example, the Air Force limits its concept of modularity, to the classical approach of functional modularity, while completely ignoring the concept of information hiding during the design of modules. This concept of information hiding, should be included so the benefits from its approach of anticipating changes to the software can be derived, and therefore increasing maintainability.

Although the Air Force has developed a method to provide creditable evaluation results, changes need to be made. Their goal is to achieve a statistical confidence level for the test data to provide a measure of software maintainability. By using multiple evaluators, it provides a broader sample size. However, the scoring technique used is still subjective in nature.

The grading criteria of the six-point system, especially the threshold and goal limits need to be statistically validated. An informal interview with a member of the staff at the Computer/Support Systems Division located at Kirtland Air Force Base, revealed that the point system originated from an attempt to achieve scores that would take the shape of a bell curve with the threshold being one standard deviation below the mean and the goal one standard deviation above the mean. A research project using the Air Force's methodology to evaluate software that possesses varying degrees of maintainability as measured by the cost and effort required by maintenance programmers to make changes is needed.

D. BEBUGGING

In an attempt to measure maintainability, rather than specifying techniques contractors were to use, the idea of "bebugging" was originated. It is a concept first proposed by Mills as an attempt for establishing confidence levels for the number of errors in a program, how long it would take to find them and what impact they would have on software reliability. The method is based on the intentional and random emplacement of errors within a program and subsequent debugging. The method also goes by such names as "inspection statistics" and "artificial bug insertation".

Gilb [13] has used the bebugging concept to measure if design specifications have been met by contractors. An example he uses to explain this concept is to consider an original design specification that called for 95 percent of all program bugs, to be successfully repaired within one hour. One hundred bugs would than be randomly inserted into the program by an independent party. A qualified maintenance programmer would then try to detect and correct the errors. If the results showed that the maximum time for repair of the 95 most quickly repaired bugs was 50 minutes, the average repair time was 30 minutes and the worst single repair time was 10 hours, then he would consider the program maintainable and meeting specifications.

To make the artificial bug insertion representative for his project a manager could choose between two approaches. The first approach as described by Fagan [114] is to take representative samples based on a proportional representation of errors. The second approach is to make the assumption that bugs can be caused by any type of programming statement. One would then insert artificial bugs according to the frequency of the types of statement.

Software contractors originally objected to this concept because they did not know if their programmers were making maintainable programs. However, they discovered that knowing that this form of testing would be used as part of the acceptance process, the programmers began to write extensive comments something that had not occurred before.

Although the literature contains few examples where this concept was applied, it has been used successfully on a Scandinavian Bank on line system and a remote job entry and multiterminal software for a micro computer system. The contractor of the system would perform the test on the first module completed and, it it passed, the contractor was on the right track. If not, he had an opportunity to change the programming and documentation methods prior to completion of the program.

Although the "bebugging" method has been used prior to completion of the project, it still requires that some coding be completed first. Thus, it is not readily extendable to be utilized during reviews conducted prior to coding. It assumes that if the errors can be found and corrected, good documentation exists. The weaknesses of this approach are:

1. It does not determine if the logic of the program is correct.

2. It does not determine if the user's requirements are met.

3. It does not address issues of adaptive or preventive maintenance. It addresses only corrective maintenance.

Even though "bebugging" has showed that it is capable of motivating a contractor or a programmer to document more, its use is limited because it does not consider if the documentation had been designed.

E. PROPOSAL

To alleviate the weaknesses of the Air Force Evaluator's Handbook and the Bebugging method, this thesis proposes a new evaluation approach. The approach consists of combining the format of the Air Force over the life cycle of the project with the means of measuring results, by inserting the use of "bebugging" within the contract.

Realizing various life cycle models exist, this thesis will follow the model prepared by the Rome Air Development

Center to provide a framework for the questions. The reason for this is because of the relationship that already exist within this model between the life cycle and the technical reviews.

The questions will cover that portion of the life cycle that explicityly pertains to the design aspect of a program. These are the Systems Requirement Review, Systems Design Review, Preliminary Design Review and the Critical Design Review.

The questions will be presented in the following manner:

1. ____ Review.
2. Purpose of the Review.
3. List of questions to be asked at the review.
4. Explanation (for applicable questions).

1. System Requirements Review

The purpose of reviewing documentation during the System Requirements Review is to determine if the Requirements Document had been developed in a formalized manner. The reason for this formalization, as explained by Quade [115] is that in the past, the Requirements Document was normally produced in a adhoc manner blending some principles of system analysis and common sense. According to Bell and Thayer [116] this first review of the Requirement Specifications "will find from one to four non trivial errors per page." This exemplifies why the aspects of documentation should be examined at this time.

The following set of questions will try to determine if the requirements and performance goals have been adequately documented. It will attempt to do this by asking questions that are related to the principles presented earlier by Heninger.

QUESTION: Was the Requirements Document designed by stating questions before trying to answer them?

EXPLANATION: If the formulation of questions is not considered first, it has been found that the available material prejudices the requirement specification so that only the easily answered questions are formulated. [60] The following table illustrates a number of topics and questions that should be asked at this time.

QUESTION: Was the Requirements Document designed by separating concerns?

EXPLANATION: This principle serves the objectives of making the document easy to change, since it causes changes to be well confined. An example considering hardware interfaces is used to explain this point. Hardware interfaces would be described without making any assumptions about the purpose of the program; meaning that the hardware section would remain unchanged if the behaviour of the program changed. The software behaviour is also described without any references to the details of the hardware devices; thus the software section would not change if the hardware changed [60].

TABLE V-I. A-7E REQUIREMENTS TABLE OF CONTENTS [60].

Table of Contents

Chapter	Contents
0 Introduction	Organization principles; abstracts for other sections; notation guide
1 Computer Characteristics	If the computer is predetermined, a general description with particular attention to its idiosyncrasies; otherwise a summary of its required characteristics
2 Hardware Interfaces	Concise description of information received or transmitted by the computer
3 Software Functions	What the software must do to meet its requirements, in various situations and in response to various events
4 Timing Constraints	How often and how fast each function must be performed: This section is separate from section 3 since "what" and "when" can change independently.
5 Accuracy Constraints	How close output values must be to ideal values to be acceptable
6 Response to Undesired Events	What the software must do if sensors go down, the pilot keys in invalid data, etc.
7 Subsets	What the program should do if it cannot do everything
8 Fundamental Assumptions	The characteristics of the program that will stay the same, no matter what changes are made
9 Changes	The types of changes that have been made or are expected
10 Glossary	Most documentation is fraught with acronyms and technical terms. At first we prepared this guide for ourselves; as we learned the language, we retained it for newcomers.
11 Sources	Annotated list of documentation and personnel, indicating the types of questions each can answer

QUESTION: Was the Requirements Document designed to be as formal as possible?

EXPLANATION: Try to avoid prose at this time so that information can be presented in as a concise and consistent manner as possible.

The following set of questions will try to determine if the documentation had been designed with the idea of stating the performance goals.

QUESTION: Is there a clear statement of performance goals in the User Requirements Statement?

EXPLANATION: This question will attempt to solve the problem of past software projects concerning the lack of adherence to stated software performance objectives over the software's life cylce.

QUESTION: Does the Requirements Document appear that it will support traceability?

2. Software Design Review

Before proceeding to the phase of software design a project manager would like to have a complete, validated and machine-independent specification of software requirements. This is what the systems requirements questions attempted to perform. However, the requirements are not really validated according to Boehm [4] "until it is determined that the resulting system can be built for a reasonable cost - and to do so requires developing one or more software designs."

Therefore the purpose of reviewing documentation at this time occurs because the development of the designs is a manual operation and most software errors are made during this phase.

The following set of questions are to determine if the proposed design has considered specific topics that are important to this particular phase of the project.

QUESTION: Does the documentation state that there has been early attention applied to the critical issues of integration and interfacing?

EXPLANATION: This is normally accomplished through a top-level expression of a hierarchial control structure routine calling an "input" and an "output" and proceeds to iteratively refine each successive lower-level component until the entire system is specified.

QUESTION: Are there graphical representations of the system embodied in the documentation at this time?

EXPLANATION: The design should be represented by an acceptable convention such as flow charts, Hipo diagrams, decision-matrix tables or a combination thereof.

QUESTION: Have each function been identified and isolated into a separate module?

EXPLANATION: The purpose of this question is to try and capture the guidelines of modularization. There are many ways to modularize and the view of Parnas [30] as presented earlier is one means which would be acceptable.

QUESTION: Does a set of simple automated consistency checks exist to validate this documentation against itself and preceeding documentation?

EXPLANATION: When manually designing a system it is difficult to keep the design consistent. Therefore it is advantageous to have a means to perform simple consistency checking. Boehm, McLean and Urfig [117] state that the use of simple consistency checking can catch dozens of potential problems in a large design specification.

The following set of questions will be used to insure that there is consistency and traceability between the Requirements Specification and Design Specification. For this reason an explanation is assumed not to be needed.

QUESTION: Can each design specification be traced to one or more of the user requirements specifications?

QUESTION: Are the performance goals the same now as they were during the requirements statement? If not why not?

QUESTION: Can the performance goals be traced to the user requirements?

3. Program Design Review

The purpose of reviewing documentation at this time is to analyze the design to determine if the proposed implementation is capable of meeting specified performance, design and verification requirements [32].

An important piece of documentation in this phase is the Computer Program Development Plan. This plan is to provide a top-level overview of the organizational responsibilities, project phasing, and major tasks to be accomplished during the software development process. A summary of the reliability technical approach, the organizational relationship, and the top level schedules for test and integration activities should be included.

The following set of questions are used to determine if a Computer Program Development Plan exists and if so, does it provide the proper items. No explanation is given as each question relates to the purpose of the development plan presented above.

QUESTION: Is there a Computer Program Development Plan?

QUESTION: Does the Computer Program Development Plan outline a schedule of tasks in chronological order?

QUESTION: Does the Computer Program Development Plan outline the functions of the program?

QUESTION: Does the Computer Program Development Plan precisely delineate the interface(s) between the program module(s)?

QUESTION: Does the Computer Program Development Plan show the structure of the data flows through the program?

The following set of questions will be used to insure that there is consistency and traceability between the requirements specification, design specification and program design. For this reason an explanation of each question is not given.

QUESTION: Can the documentation be traced from its present state to the design specification and then to the requirements specification?

QUESTION: Are the performance goals the same now as they were during the design specification and requirement specification stages?

QUESTION: Can the performance goals be traced to the User Requirement Document?

4. Critical Design Review

The purpose of reviewing documentation at this time is because it is the last chance to correct design flaws before coding begins.

An important piece of documentation in this phase is the Computer Program Test Plan. It covers planning and scheduling of formal software verification through qualification testing. If the development effort output consists of more than one identifiable computer program, the span of the test plan should include formal qualification testing of the entire software system.

The following set of questions are used to determine if a Computer Program Test Plan exists and if so does it entail all of the proper topics. No explanation is given as each question relates to the purpose of the Program Test Plan presented above.

QUESTION: Does a proposed user manual exist?

QUESTION: Does a first draft of the maintenance manual exist?

QUESTION: Does a proposed acceptance test plan exist?

QUESTION: Does there exist a detailed breakdown of the major functions?

QUESTION: Does there exist detailed algorithms for each module?

QUESTION: Does there exist a detailed description of the interfaces?

QUESTION: Does there exist a detailed description of the data structure?

The following set of questions will be used to insure that there is consistency and traceability between the requirements specification, design specification, program design and critical design. For this reason an explanation of each question is not given.

QUESTION: Can the documentation be traced from its present state to the requirement specification?

QUESTION: Are the performance goals the same now as they were during the requirement specification?

QUESTION: Can the performance goals be traced to the user requirement document?

5. Additional Questions

The following set of questions and explanations are included to show the type of question that can be asked to bring the Air Force questionnaires up-to-date with concern to the latest concepts in software engineering.

QUESTION: Does the documentation include formal specifications?

EXPLANATION: Formal specifications are needed:

1. To describe the problem to be solved.
2. For communication between software engineers.

3. To free the programmer from needing to know how the rest of the system works.

4. To support the development of multi-version software.

5. To complete the description of the design decisions.

6. To permit verification of intermediate design decisions.

QUESTION: Does the documentation present the specification as precise?

EXPLANATION: If the specification is not precise then the following will occur:

1. The problem solved may not be exactly that whose solution is needed.

2. Cooperating software engineers may develop programs that are not compatible.

3. The programmer may have to study other people's programs in order to determine exactly what is required of his program.

4. Later refinement of a program may not be consistent with the design decisions that were the intention of those who wrote the program.

5. The interpretation of the specification that was used in verifying the correctness of a design may not be the same as the interpretation made by the implementator.

QUESTION: Does the documentation include a separate section for the description of the abstract interface module(s)?

EXPLANATION: The abstract interface(s) should be defined by writing down all assumptions about the interface and note which will change and which will not.

QUESTION: Does the documentation explicitly state that the interface module does not include programs that are device dependent?

EXPLANATION: If programs are included in the device interface module that are device dependent, the device interface module may need to be changed if the device is changed.

QUESTION: Does the documentation include a separate section for the description of an interface?

EXPLANATION: An interface between two programs is defined by the set of explicit and implicit assumptions they name about each other.

6. Conclusions

Although these questions are not all encompassing they have been designed to take the place of requirements-design-code consistency checking and automatic programming, which have not been developed yet. They will undoubtedly also help ensure that the latest software engineering techniques are used, along with improving software productivity and quality in the area of management. The largest software management areas that will be improved are:

PLANNING: Large amounts of effort and time will not be wasted because of tasks that are no longer unnecessarily performed, or poorly synchronized.

CONTROL: Plans will be forced to be kept up-to-date and used to manage.

SUCCESS CRITERIA: Emphasis will now be placed on the activities of requirement and design validation, test planning and drafting of user documentation [10].

VI. CONCLUSIONS AND RECOMMENDATIONS

The software engineering literature is replete with proposals of various tools that claim to offer solutions to many of the problems associated with software development. Successful small projects are frequently cited by the authors of these books and articles to support their claim that the success of these projects can be directly attributed to the use of the particular tool or method being advocated. Similar glowing reports on successful large scale projects are noticeably scarce. It may well be that the success of these small projects are due more to their size than to the tools or methods being promoted. While this may give further credence to the "divide and conquer" approach to software development, further research is required in order to determine the usefulness of some of the tools in projects of greater magnitude. Organizations such as the Office of Naval Research (ONR) and the Naval Postgraduate School possess the necessary facilities and technical expertise to perform research, similar to the A-7 Operational Flight Program project mentioned previously, to validate the utility of the proposed tools and techniques in large scale development projects as well as provide useful models that may be utilized by project managers in software acquisition projects. The emphasis of this research should be directed towards developing the means to automate the validation and verification of the specification and design phases of software development. The reason for this emphasis is two-fold. First, as was graphically illustrated in Chapter IV, these are the phases where the majority of errors can be traced. Furthermore, since during these phases there are fewer ramifications that result from changes than occur in the latter phases of development, the cost of correcting these errors early is, similarly, less costly. The second reason for emphasizing this area of research is that, since DoD organizations often issue contracts for software development rather than utilize inhouse resources, the tools that apply to the earlier phases of the life cycle hold promise for the greatest return on the research investment.

Recognizing, however, that the automation of this process will require years of research before the automated tools and techniques will be available for wide scale use, non-automated means of assuring software quality must also be explored. The inevitability of changing user's requirements as well as the realization that the development of error-free software is beyond the current state-of-the-art, ensure that the maintainability of the software must be a primary consideration in any development project. The approach offered by the U.S. Air Force is one that is currently available to aid the project manager in measuring the maintainability of the delivered product. One advantage of this approach is that it may be used as part of the technical review and audit process to identify potential problems affecting the maintainability early in the development phases by examining the documentation used at these reviews as well as a measurement technique for the acceptance testing and evaluation processes. Unfortunately, there has been no empirical evidence offered that statistically validates the current rejection criteria used in this evaluation method. Further research is required to empirically validate this approach. One stumbling block to this research is the lack of revelant maintenance data on existing software systems that could provide a data base for this research. The GAO study [5] on software maintenance management practices cited a lack of a uniform definition of what legitimately constitutes software maintainance activities as a primary cause for not having appropriate data available. It is, therefore, recommended that a directive be issued that contains a DoD definition of maintenance, along with the lines of Swanson's recommendation as discussed in Chapter III. It should also require that a record of the time and effort devoted to the maintenance of existing software be kept by the user organizations. This will serve to both focus management attention on this vital area as well as provide a statistical data base to support research.

Due in large measure to the dual communities that have evolved as a result of the Brook's Act separating the development and acquisition practices associated with non-tactical (ADP) software from those applicable to tactical or embedded computer software, there exists several duplicated and even conflicting directives and instructions. While recognizing that the acquisition process is, by law, different, the issues surrounding the development of high quality software are not. One step in reducing this multiplicity of guidance would be to adopt a single software development standard that would be applicable to both communities. MIL-STD-1679 represents a good starting point for this unified standard. More emphasis on ensuring the maintainabiity of the delivered product is necessary. Specific requirements for the evaluation of maintainability as part of the design review and acceptance testing processes should be included in this unified standard. Both the Air Force and "bebugging" techniques may be valid means of accomplishing this requirement, although not necessarily the only ones. The issue of requiring regression testing should also be addressed in this new, unified instruction.

Finally, the development of quality software requires more than just tools and methods. Another vital ingredient is the presence of highly motivated, technically proficient managers who are familiar with these tools and can integrate them into projects where their use is justified. The Computer Systems Management curriculum at the Naval Postgraduate School is one DoD effort to provide the necessary personnel to staff various software development projects. Further research is also required to determine the feasibility of creating either a separate staff corps or a subspeciality within the Engineering Duty Officer community, such as is done in the other services. The creation of a separate career path for software engineer and management specialists would provide the Navy with a cadre of professionals who would be able to stay current with this rapidly expanding field as well as provide the technical expertise required to ensure the success of software acquisition and development projects.

APPENDIX A

This appendix contains the 83 questions utilized by evaluator teams at the U.S. Aiɪ Force Test and Evaluation Center, Computer Support Division, Kirkland Air Force Base, New Mexico. Used to evaluate software documentation for six aspects of maintainability, these questions are reproduced from Reference [112].

QUESTION DATA SHEET

Question Number D-1

QUESTION: The documentation includes a separate part for the description of external interfaces.

CHARACTERISTIC: Modularity (format modularity).

EXPLANATIONS: Personnel working in functional areas need to have information available in one place.

EXAMPLES: An Interface Control Document (ICD). An operator's Manual.

GLOSSARY:

Part: Section, volume, document, subsection, etc. as appropriate.

External interfaces: Program input and output data, interrupts.

SPECIAL RESPONSE INSTRUCTIONS:

Answer A if one separate part exists.

Answer B-E if the external interfaces are described in several separate parts depending upon the effectiveness of that distribution.

Answer F if no description of external interfaces is available.

QUESTION DATA SHEET

Question Number D-2

QUESTION: The documentation includes a separate part for the description of each major function.

CHARACTERISTIC: Modularity (format modularity).

EXPLANATIONS: Personnel working on a specific function should have all relevant information available in one piece.

EXAMPLES: Personnel working only on the navigation function of an aircraft operational flight program should have navigation functional descriptions in one place.

GLOSSARY:

Part: Section, volume, document, subsection, etc. as appropriate.

Major function: As defined by the overview or other equivalent information: may be a component, module, etc.

SPECIAL RESPONSE INSTRUCTIONS:

Answer A if a separate part exists for each major function.

Answer B - E if each major function is described in several separate parts depending upon the effectiveness of that distribution.

Answer F if there is no description of each major function.

QUESTION DATA SHEET

Question Number D-3

QUESTION: The documentation includes a separate part for the description of the program global data base.

CHARACTERISTIC: Modularity (formal modularity).

EXPLANATIONS: Personnel working with the data base should have a description of all global data items in one place.

EXAMPLES: There should be a separate part of documentation containing descriptions, types, ranges, sizes, formats, etc. of the global data items. Where lists are not complete, plans for completion should be evident.

GLOSSARY:

Part: Section, volume, document, subsection, etc. as appropriate.

Global data base: Set of all variables, constants, etc. which can be accessed by more than one program module: e.g., FORTRAN's COMMON, JOVIAL's COMPOOL, assembly's DATA MODULE, etc.

SPECIAL RESPONSE INSTRUCTIONS:

Answer A if a separate part exists or there is clearly no global data.

Answer B - E if the global data base is described in several separate parts depending upon the effectiveness of that distribution.

Answer F if no description of the global data base exists.

QUESTION DATA SHEET

Question Number D-4

QUESTION: Major parts of the documentation are essentially self-contained.

CHARACTERISTIC: Modularity (format modularity).

EXPLANATIONS: Sampling major parts of the documentation for the amount of cross referencing and the essential nature of the cross referencing should give the evaluator a general impression as to level of agreement/disagreement with the question statement. However, cross referencing for the purpose of eliminating bulky redundancies is accepatable.

EXAMPLES:

GLOSSARY:

Major parts: as essentially defined by documentation table of contents and physical structure (volumes, sections, units, etc.): might include major functions, data base description, external interfaces, test plan, conventions and standards, etc.

Self-contained: Independent, stand-alone document. Makes no cross references to other major parts of the documentation.

SPECIAL RESPONSE INSTRUCTIONS:

AD-A120 423 SOFTWARE ENGINEERING BASICS: A PRIMER FOR THE PROJECT MANAGER (U)
STEVEN PATRICK ARTZER, ET AL NAVAL POSTGRADUATE SCHOOL, MONTEREY, CA JUN 82

UNCLASSIFIED

3 OF 3
ADA
120423

END
DATE
FILMED
4-28-83
NTIS

QUESTION DATA SHEET

Question Number D-5

QUESTION: The documentation has been physically separated into (sets of) volumes each with a distinct purpose.

CHARACTERISTIC: Modularity (format modularity).

EXPLANATIONS: Each (set of) volume's introduction should include an indication of what the (set of) volume's purpose is. A brief scan of the document should give the evaluator a general impression of whether that purpose is relatively distinct, mixed, matches the stated purpose, etc. Each physically separate volume should be checked and an accumulative impression formed of the level of agreement/disagreement with the question statement.

EXAMPLES: Maintenance information should not be physically included in an operator's handbook.

GLOSSARY: Distinct purpose: These might include functional specification, detailed specification, maintenance manual, user's guide, data base description, problem reports, installation instructions, documentation plan, test plan, etc.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-6

QUESTION: The documentation indicates that each global data structure is partitioned into functionally related sets of variables.

CHARACTERISTIC: Modularity (data modularity).

EXPLANATIONS: Documentation describing the program global data base should include the set of all global data and how it has been partitioned into global data structures.

EXAMPLES: Geodetic site data should be grouped in one global data structure.

GLOSSARY:

Global data: Any variable or constant which can be accessed by more than one module of a program.

Global data structure: A particular grouping of global data variables and/or constants; e.g., FORTRAN's COMMON, JOVIAL's COMPOOL.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if there is no global data (hence no global data structures); the implication is that data coupling is decreased if there is no global data.

QUESTION DATA SHEET

Question Number D-7

QUESTION: The documentation indicates that data storage locations are not used for more than one type of data structure.

CHARACTERISTIC: Modularity (data modularity).

EXPLANATIONS: Typical multiple use of data storage locations would be dynamic memory management schemes, equivalence and overlays. Program documentation (perhaps at the module level) should describe any use of storage locations for these types of uses.

EXAMPLES: Any use of the EQUIVALENCE statement in FORTRAN, or SAME SORT, SAME AREA, or REDEFINE in COBOL shold be documented.

GLOSSARY: Type: Examples of types of data structures would be integer, real, character, array of integer, array of real, array of characters, records, files, etc.

SPECIAL RESPONSE INSTRUCTIONS: If not indicated in the documentation either way, answer D.

QUESTION DATA SHEET

Question Number D-8

QUESTION: The program control flow is organized in a top down hierarchical tree pattern.

CHARACTERISTIC: Modularity (processing modularity).

EXPLANATIONS:
The documentation should include a program overview section which will likely describe the overall program control flow among modules in narrative or chart form.
Control paths which are not strictly down or up in the sense of level tend to detract from modularity because of the associated lack of independence which is introduced.

EXAMPLES:

GLOSSARY: Top down hierarchical tree pattern: One imagines a root system of a tree with each junction (node) representing a major program function or module and each branch a control path between the nodes.

SPECIAL RESPONSE INSTRUCTIONS: Answer F if there is no description (narrative or chart) of overall program control flows.

QUESTION DATA SHEET

Question Number D-9

QUESTION: The documentation indicates that program initialization processing is done by one (set of) module(s) designed exclusively for that purpose.

CHARACTERISTIC: Modularity (processing modularity).

EXPLANATIONS:
The documentation describing the program functions and control flow should also describe how the initial program state is determined (e.g., via execution of one initialization module or not). Checks of module processing may indicate whether any initialization processing is mixed with other application functions.

It is usually better if each function, module, submodule, etc. does not handle its own global initialization. There should be one part (e.g., a few modules) of the program which is for initialization.

EXAMPLES:

GLOSSARY: Initialization: The preparatory steps required to set the initial program data and control states.

SPECIAL RESPONSE INSTRUCTIONS: If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for initialization, then strong agreement with the question should be so indicated. If in addition, these initialization modules are all executed in preparation to any other functional activity as the first program action, then there should be essentially complete agreement with this question's statement.

QUESTION DATA SHEET

Question Number D-10

QUESTION: The documentation indicates that program termination processing is done by one (set of) module(s) designed exclusively for that purpose.

CHARACTERISTIC: Modularity (processing modularity).

EXPLANATIONS:

The documentation describing the program functions and control flow should also describe how the final program state is determined (e.g., via execution of one termination module or not). Checks of module processing may indicate whether any termination processing is mixed with other application functions.

It is best if each function, module, submodule, etc. does not handle its own termination. There should be one part (e.g., a few modules) of the program which is for termination processing.

EXAMPLES: FORTRAN's STOP statement and COBOL's STOP RUN statement could be within the processing area.

GLOSSARY: Termination: The terminal steps required to set the final program data and control states (might be due to normal/abnormal termination).

SPECIAL RESPONSE INSTRUCTIONS: If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for termination processing, then strong agreement with the question should be so indicated. If in addition, these termination modules are executed only as a systematic program termination procedure, then there should be essentially complete agreement with this question's statement. Variations on the program termination processing should result in appropriate variations in the evaluator response depending on how much termination processing is mixed with other application functions.

QUESTION DATA SHEET

Question Number D-11

QUESTION: The documentation indicates that program I/O is done by one (set of) module(s) designed exclusively for that purpose.

CHARACTERISTIC: Modularity (processing modularity).

EXPLANATIONS:

The documentation describing the program functions and control flow should also describe how the program I/O is done (e.g., via execution of one module or more). Checks of module processing may indicate whether any I/O functions are mixed with other application functions.

It is best if each function, module, submodule, etc. does not handle its own I/O. There should be one part (e.g., a few modules) of the program which is for I/O processing.

EXAMPLES:

GLOSSARY: I/O: Input or output of program data.

SPECIAL RESPONSE INSTRUCTIONS: If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for I/O, then appropriate agreement with the question should be so indicated. Variations on the program I/O processing should result in appropriate variations in the evaluator response depending on how much I/O is mixed with other application functions.

QUESTION DATA SHEET

Question Number D-12

QUESTION: The documentation indicates that program error processing is done by one set of modules designed exclusively for that purpose.

CHARACTERISTIC: Modularity (processing modularity).

EXPLANATIONS:
The documentation describing the program functions and control flow should also describe how the program processes any error condition (e.g., via execution of one error processing module or not). Checks of module processing may indicate whether any error processing functions are mixed with other application functions.

It is best if each function, module, submodule, etc. does not handle its own error processing unless adequate corrective measures are appropriate. There should be one part (e.g., a few modules) of the program which is for error processing.

EXAMPLES: Editing of input data should be documented.

GLOSSARY: Error processing: The steps required to set program data and control states following the detection of an error condition.

SPECIAL RESPONSE INSTRUCTIONS: If the partitioning is such that each function is performed by a set of modules and there is one module in each set expressly for error processing, then appropriate agreement with the question should be so indicated. If in addition, these error processing modules are systematically organized as an error processing function, then there should be essentially complete agreement with this question statement.

QUESTION DATA SHEET

Question Number D-13

QUESTION: Each physically separate part of the documentation includes a useful table of contents.

CHARACTERISTIC: Descriptiveness (format descriptiveness).

EXPLANATIONS: Each separately bound part of the set of documentation for this program has its own table of contents to assist in locating program information.

EXAMPLES:

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Answer A to F depending upon the percentage of documents which have a useful table of contents: e.g., 100% - answer A, 0% - answer F.

QUESTION DATA SHEET

Question Number D-14

QUESTION: Each physically separate part of the documentation includes a useful glossary of major terms and acronyms unique to that document.

CHARACTERISTIC: Descriptiveness (format descriptiveness).

EXPLANATIONS: Each separately bound part of the set of documentation has its own glossary of major terms and acronyms to assist in clarifying other documentation.

EXAMPLES:

GLOSSARY:
Acronym: A term formed by the initial letter(s) of a series of one or more words.
Example: FORTRAN = FORmula TRANslation

SPECIAL RESPONSE INSTRUCTIONS: Answer A to F depending upon the percentage of documents which have a useful glossary: e.g., 100% - answer A, 0% - answer F.

QUESTION DATA SHEET

Questi⌐ Number D-15

QUESTION: Each physically separate part of the ·ocumentation includes a useful index.

CHARACTERISTIC: Descriptiveness (format descri. .veness).

EXPLANATIONS: Each separately bound part of the set of documentation has its own index to assist in locating information.

EXAMPLES:

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Answer A to F depending upon the percentage of documents which have a useful index: e.g., 100% - answer A, 0% - answer F.

QUESTION DATA SHEET

Question Number D-16

QUESTION: It is easy to locate specific information within the documentation.

CHARACTERISTIC: Descriptiveness (format descriptiveness).

EXPLANATIONS: The evaluator should repeatedly conceptualize the need for locating a specific piece of information that might be needed for maintenance, and then check the documentation for the effort required to locate the information.

EXAMPLES: One piece of frequently needed information might be the contents of the parameter list. Another might be a list of what modules call another module. The evaluator should consider some specific piece of information and assess the ease of locating that information.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-17

QUESTION: The documentation includes a useful version description document.

CHARACTERISTIC: Descriptiveness (format descriptiveness).

EXPLANATIONS: Some document should be readily available which describes the current operational version of each configuration controlled program. In hierarchical library systems, documents should be available describing each level of the library.

EXAMPLES:

GLOSSARY: Version description document: A document which describes the version of each computer program.

SPECIAL RESPONSE INSTRUCTIONS: If the documentation is a baseline (original version or an all-encompassing rewrite), the evaluator should answer A.

QUESTION DATA SHEET

Question Number D-18

QUESTION: A useful master list is available which identifies all software documentation.

CHARACTERISTIC: Descriptiveness (format descriptiveness).

EXPLANATIONS: A reference list should be available in one overview document or in a separate document which lists at least all delivered program-related documentation by name and description; if several programs are part of the software developement effort, then the list should contain information on all programs.

EXAMPLES:

GLOSSARY: Master list: This may be a reference list in one overview document, or a spearate document itself.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-19

QUESTION: Any dynamic allocation of resources (storage, timing, priority, hardware services, etc.) is explained in the documentation.

CHARACTERISTIC: Descriptiveness (constraints descriptiveness).

EXPLANATIONS: The documentation describing the functional/ detailed program specifications should include a section which explains what dynamic allocation features are used by the program. These features may be considered necessary depending upon the application, but all are considered to increase the effort required to maintain a program.

EXAMPLES: The most common dynamic allocation feature is probably storage allocation. There may be allocation routines which must be called to get or release memory. Also, the priority scheme or timing allocation for particular "rate" groups may depend upon certain phases of a mission and dynamically change on that basis. Likewise assignment of control of tape drives, discs, communication lines or other hardware may be done in some dynamic manner.

GLOSSARY: Dynamic allocation: Any assignment of a resource which is (or can be) done during the execution of a program; contrasts with "static allocation" which implies the resource assignment remains fixed throughout program execution.

SPECIAL RESPONSE INSTRUCTIONS:

Answer A if it is clear there is no dynamic allocation of resources for this program.

Answer B - F otherwise.

QUESTION DATA SHEET

Question Number D-20

QUESTION: Timing requirements for each major function of the program are explained in the documentation.

CHARACTERISTIC: Descriptiveness (constraints descriptiveness).

EXPLANATIONS: The allocated time for each major function operating in a real-time environment should be described in the documentation. In addition, the timing relationships among major functions, or the framing scheme, should also be described and readily available.

EXAMPLES:

GLOSSARY: Major function: The program overview, hierarchical chart, etc. will ordinarily define what major function (and its components) means; it usually will correspond to a module or group of modules as defined for a given program evaluation.

SPECIAL RESPONSE INSTRUCTIONS:
Answer A if it is clear that this program has no timing requirements (e.g., is non-real time).
Answer B - F otherwise.

QUESTOIN DATA SHEET

Question Number D-21

QUESTION: Storage requirements for each major function of the program are explained in the documentation.

CHARACTERISTIC: Descriptiveness (constraints descriptiveness).

EXPLANATIONS: Allocated storage requirements for each major function should be described in the documentation. Even if a program does not have any "critical" storage requirements, there should be an explanation in the documentation covering the program's environment.

EXAMPLES:

In a program operating in a paged storage environment, the page limitations (number of pages, boundary requirements, etc.) should be described.

For programs operating in a resident/non-resident environment, relationships to the roll-in area requirements should be described.

GLOSSARY: Major function: The program overview, hierarchical chart, etc. will ordinarily define what major function (and its components) means; it usually will correspond to a module or group of modules as defined for a given program evaluation.

SPECIAL RESPONSE INSTRUCTIONS: Answer F if there is no explanation of the storage requirement(s).

QUESTION DATA SHEET

Question Number D-22

QUESTION: The inputs to each module are explained in the documentation.

CHARACTERISTIC: Descriptiveness (module descriptiveness).

EXPLANATIONS: Input parameters passed via parameter packages or argument lists and global data used by each module as input should be described.

EXAMPLES: The documentatin for a trigonometric subroutine describes what data is input (an angle), the form (in radians), limitations ($0 \leq$ angle $\leq \pi/2$), and how it is input passed as a single precision real number in the first parameter).

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-23

<u>QUESTION</u>: The processing done by each module is explained in the documentation.

<u>CHARACTERISTIC</u>: Descriptiveness (module descriptiveness).

<u>EXPLANATIONS</u>: The algorithm(s) which generate the outputs from the inputs should be described in the documentation.

<u>EXAMPLES</u>: A trigonometric function should have a description of the function itself, the algorithm used, and any limitations.

<u>GLOSSARY</u>:

<u>SPECIAL RESPONSE INSTRUCTIONS</u>:

QUESTION DATA SHEET

Question Number D-24

QUESTION: The outputs from each module are explained in the documentation.

CHARACTERISTIC: Descriptiveness (module descriptiveness).

EXPLANATIONS: Output parameters passed via parameter or argument lists and global data altered by each module should be described.

EXAMPLES: The documentation for an inverse trigonometric subroutine describes what data is output (an angle), the form (in radians), and how it is output (passed as a double precision real number in the second parameter).

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-25

QUESTION: Special processing considerations (error, interrupt, etc.) of each module are explained in the documentation.

CHARACTERISTIC: Descriptiveness (module descriptiveness).

EXPLANATIONS: Any special considerations, such as the different types of errors possible and their effects, the effects of interrupts and the effects of other asynchronous events should be described.

EXAMPLES: In a message processing program, processing limitations may cause loss of an incoming character. The documentation for the input handler should describe this condition and its response to it.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-26

QUESTION: There is a flow chart (or equivalent) for each module which adequately illustrates the inputs, general processing, and outputs for the module.

CHARACTERISTIC: Descriptiveness (module descriptiveness).

EXPLANATIONS: Some form of functionally-oriented presentation of each of the program components should be available in the documentation. This could take the form of flowcharts, Process Design Language (PDL) with functional commentary, Hierarchical Input-Processing-Output (HIPO) charts, etc.

EXAMPLES:

GLOSSARY: Flowchart (or equivalent): A logic flow diagram in which symbols are used to represent operations, data, flow, equipment, etc. Examples are: FORTRAN flowchart, Process Design Language (PDL), Hierarchical Input-Processing-Output (HIPO) chart, etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer F if module flowcharts (or equivalent) do not exist.

QUESTION DATA SHEET

Question Number D-27

QUESTION: Program initialization and termination processing is explained.

CHARACTERISTIC: Descriptiveness (external interface descriptiveness).

EXPLANATIONS: The documentation should cover both the data and the steps required to initialize the operation of this program within the system and to effect both normal and abnormal termination of the program.

EXAMPLES: An Operational Flight Program may have no termination procedures short of power off; however, most such programs determine the source of the power outage, and whether any memory locations need be protected, etc. Such considerations should be documented.

GLOSSARY:

Initialization: The preparation steps required to set the initial program data and control states.

Termination: The terminal steps required to set the final program data and control states (might be due to normal/abnormal termination).

SPECIAL RESPONSE INSTRUCTIONS: The evaluator should study both initialization and termination processing explanations. The response A-F should reflect overall how well both have been explained.

QUESTION DATA SHEET

Question Number D-28

QUESTION: Recovery from externally generated error conditions which could affect the program is explained.

CHARACTERISTIC: Descriptiveness (external interface descriptiveness).

EXPLANATIONS: The documentation should include an explanation of overall error processing. This description should include a description of the recovery of the program from error conditions generated external to the program, but affecting its capability to function. In most cases, this explanation will concern the recovery from lack of or bad input data or parameters to the program.

EXAMPLES:

GLOSSARY: Recovery: The procedures taken to report/correct some program failure (resulting from an external error condition in this case and probably recognized as bad input data).

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-29

QUESTION: The process of recovering from internally generated error conditions is explained.

CHARACTERISTIC: Descriptiveness (external interface descriptiveness).

EXPLANATIONS: The documentation should include an explanation of overall error processing. This description should include a description of the recovery of the program from error conditions encountered within the program and not directly caused by the environment external to the program.

EXAMPLES: The documentation explains that, in cases where a divide by zero is possible, a check is made of the divisor and alternate processing is instituted to recover from the error.

GLOSSARY: Internal error condition: Any algorithm failure due to processing of internally defined data.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-30

QUESTION: Input of program data is explained.

CHARACTERISTIC: Descriptiveness (external interface descriptiveness).

EXPLANATIONS: The documentation should describe what data is input, the form of the data, any limitations on the data, and how it is input.

EXAMPLES:

1. Card deck or card deck image input: Line-by-line description of input, giving format, range or limitations of each data field, type (numeric or alphanumeric, integer or floating point), etc.
2. Multiplex Bus: Description of all data structures to be received from the bus, giving source and timing of data blocks (such as a block received from the inertial navigation system once per second), the sequence, definition, and scale factors of the parameters in a block, etc.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-31

QUESTION: Output of program data is explained.

CHARACTERISTIC: Descriptiveness (external interface descriptiveness).

EXPLANATIONS: The documentation should describe what data is output, the form of that output, and how it is output.

EXAMPLES: Complete description of the program output, be it:
1. Listing (printout),
2. CRT display (data displayed on a Heads Up Display [HUD]), or
3. Mux Bus, etc.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-32

QUESTION: There is a useful set of charts which show the general program control and data flow hierarchy among all modules.

CHARACTERISTIC: Descriptiveness (internal interface descriptiveness).

EXPLANATIONS: Whatever method is used to present the flow, the presentation should be understandable and complete.

EXAMPLES: The documentation should include a set of system flowcharts, Process Design Language (PDL), Hierarchical Input-Processing-Output (HIPO), etc. which show the program control and data flow either together or separately.

GLOSSARY: Chart: Flowchart, Process Design Language (PDL), Hierarchical Input-Processing-Output (HIPO) chart, etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer F if no set of charts exists.

QUESTION DATA SHEET

Question Number D-33

QUESTION: There is a master list (chart, table, section, etc.) identifying where each global variable is used.

CHARACTERISTIC: Descriptiveness (internal interface descriptiveness).

EXPLANATIONS: A part of the documentation should be a master list identifying where each global variable is used. This list contains information used by maintainers of all modules and it is important they use the same list.

EXAMPLES: In many programming systems, an automated global data cross-reference report may be generated.

GLOSSARY: Global viariable: Any variable which can be accessed by more than one module of a program; global constants should also be identified.

SPECIAL RESPONSE INSTRUCTIONS: Answer F if no master list or its equivalent exists. Answer A if it is clear that no global variables exist in the program.

QUESTION DATA SHEET

Question Number D-34

QUESTION: The global variable master list includes information about each global variable such as type, range, scaling, units, etc.

CHARACTERISTIC: Descriptiveness (internal interface descriptiveness).

EXPLANATIONS: The documentation should contain a separate data base description in which all global data is described to include information on type, range, etc. This list is important in that it contains information used by maintainers of all modules.

EXAMPLES:

GLOSSARY: Global variable: Any variable which can be accessed by more than one module or a program; global constants should also be identified.

SPECIAL RESPONSE INSTRUCTIONS: Answer F if no master list or its equivalent exists. Answer A it it is clear that no global variables exist in the program.

QUESTION DATA SHEET

Question Number D-35

QUESTION: The use of any complex mathematical model (technique, algorithm) is explained in the documentation.

CHARACTERISTIC: Descriptiveness (math model descriptiveness).

EXPLANATIONS: The documentation should contain details on the use of any complex algorithm to include input requirements and limitations.

EXAMPLES: The documentation for a numerical integration algorithm might specify that a minimum number of intervals be selected for a specified result accuracy.

GLOSSARY: Complex mathematical model: e.g., Fourier transform, Laplace transform, numerical integration/differentiation scheme, control theory algorithm, statistical technique, etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if it is clear that there are no complex mathematical models (techniques, algorithms) used in the program.

QUESTION DATA SHEET

Question Number D-36

QUESTION: The documentation on each complex mathematical model includes information such as a derivation, accuracy requirements, stability considerations and references.

CHARACTERISTIC: Descriptiveness (math model descriptiveness).

EXPLANATIONS: The documentation should contain enough detailed explanation or cross-references to allow the maintainer to modify the algorithm or its implementation and be aware of the implications or be able to locate references which make the implications clear.

EXAMPLES: A numerical algorithm that depends on double precision processing should have a description of the implications to accuracy if single precision were to be substituted.

GLOSSARY: Complex mathematical model: e.g., Fourier transform, Laplace transform, numerical integration/differentiation scheme, control theory algorithm, statistical technique, etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if it is clear that there are no complex mathematical models (techniques, algorithms) used in the program.

QUESTION DATA SHEET

Question Number D-37

QUESTION: It appears that a useful set of standards has been followed for the development of the documentation.

CHARACTERISTIC: Consistency (format consistency).

EXPLANATIONS: Consistent documentation means that the maintainer can spend less time learning the organization of the documentation and more time learning the content. The documentation should be scanned for adherence to standards.

The evaluator may know in advance that documentation standards were generated and he can see that they were followed.

The evaluator may not know in advance but may be able to tell from the organization of diverse parts of the documentation that standards were available and followed.

Confusing documentation organization indicates misuse or no use of documentation standards.

EXAMPLES: Following either contractor standards developed locally or universal standards (e.g., ANSI FORTRAN) which help understandability. Usually the format consistency of the documentation indicates how much a standard/convention, etc. has been followed.

GLOSSARY: Standards: Procedures, rules, and conventions used for prescribing disciplined program design and implementation.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-38

QUESTION: It appears that a set of standards has been followed for the construction of all (program and module) flowcharts (or equivalent).

CHARACTERISTIC: Consistency (format consistency).

EXPLANATIONS: The flowcharts of the program and its modules should be scanned for conventional use of symbols, labeling consistency, etc. There may be a stated standard (e.g., ANSI FORTRAN) against which the flowcharts may be compared.

EXAMPLES: The documentation contains a section describing the flowcharting methodology and it is clear from the flow-charts that the methodology has been followed.

GLOSSARY:

Standards: Procedures, rules and conventions used for prescribing program design and implementation.

Flowchart (or equivalent): A logic flow diagram in which symbols are used to represent operations, data, flow, equipment, etc. In the broad sense, would include FORTRAN flowchart, Process Design Language (PDL), Hierarchical Input-processing-Output (HIPO), etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer F if there are no flowcharts (or equivalents).

QUESTION DATA SHEET

Question Number D-39

QUESTION: Documentation of each major functional part of the program follows the same format.

CHARACTERISTIC: Consistency (format consistency).

EXPLANATIONS: Each major functional area of a program should have the same documentation format as far as is practicable in order to aid understandability.

EXAMPLES: An airborne computer may contain major modules dedicated, for instance, to navigation, bombing, and air-to-air. Each of these modules would need input, output, and processing sections. All input sections should be similar; all output sections should be similar, etc.

GLOSSARY: Major functional part: The program overview, hierarchical chart, etc. will ordinarily define what major function (and its components) means; it usually will correspond to a module or group of modules as defined for a given program evaluation.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-40

QUESTION: The format of the documentation reflects the organization of the program.

CHARACTERISTIC: Consistency (format consistency).

EXPLANATIONS:

Program parts are easier to maintain if the documentation has separate sections to describe each of those parts. This simplifies looking for details concerning those program parts.

There can be other considerations which may influence the evaluator in responding to this question. What is desired is basically the evaluator's general impression as to the usefulness of the documentation format in understanding the overall program organization.

EXAMPLES: Major program functions, the program data base, etc. might have separate sections. The descriptions of how the program is designed to be tested should be reflected in the format of the documentation such as providing sections for unit test procedures and sample test data if applicable.

GLOSSARY: Organization of the program: Design of the program as components, modules, global data base, units, segments, etc.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-41

QUESTION: It appears that programming conventions have been established for the interfacing of modules.

CHARACTERISTIC: Consistency (design consistency).

EXPLANATIONS: Module interface design is extremely important, improper interfacing can lead to many hidden errors. Program design conve: tions hsould be documented. In addition, the module descriptions can be scanned to determine whether such conventions have been established and/or followed. The establishment of linkage conventions is especially import for assembly language modules.

EXAMPLES: Inputs and outputs, both argument type and global data type, require coordination between the sender(s) and the receiver(s). Such coordination requires explicit description of all attributes of each such variable and should be listed in an interface control document.

GLOSSARY:

Interfacing of modules: The passing of control, data, or services between two or more modules.

Convention: Agreed method or form of presentation to provide consistency and understanding.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if it is clear that there is no interfacing between any of the modules.

QUESTION DATA SHEET

Question Number D-42

QUESTION: It appears that programming conventions have been established for I/O processing.

CHARACTERISTIC: Consistency (design consistency).

EXPLANATIONS:

Program I/O processing is the interface of the program to the rest of its environment.

The module descriptions should be scanned to determine if any particular design consistency/conventions have been followed for program I/O processing.

EXAMPLES: One module or set of modules should be clearly identified as interfacing the computational system to the real world. All attributes of all inputs and all outputs should be clearly identified. This data is essential to all personnel interfacing with any I/O data, whether externally (to/from real world) or internally (to/from processing routines).

GLOSSARY: I/O processing: The physical input or output of program data.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-43

QUESTION: It appears that design conventions have been established for error processing.

CHARACTERISTIC: Consistency (design consistency).

EXPLANATIONS: Centralized processing of error conditions generally improves the maintainability of a program. Under such centralized error processing, any module which communicates an error condition to an error processing routine must do so properly. Therefore, error processing procedures must be documented and followed.

EXAMPLES: An error type is generated and passed to the error processing routine(s). The routine generating the error type "knows" that the error processing routine will handle it properly when both parties have followed the documented procedures.

GLOSSARY: Error processing: The procedure followed after a program failure due to some recognized error condition.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-44

QUESTION: A naming convention for modules appears to have been used.

CHARACTERISTIC: Consistency (design consistency).

EXPLANATIONS: Naming conventions help to describe processing and input/output. The maintenance programmer should be able to easily recognize calls to processes external to the module being changed. Although the listing may not be available to confirm conventions, the documentation should contain standards or conventions for naming yet-to-be-designed modules.

EXAMPLES: All routine names begin with "SUB" (for subroutine) or "XR" (for external-routine).

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-45

QUESTION: A naming convention for global variables appears to have been used.

CHARACTERISTIC: Consistency (design consistency).

EXPLANATIONS: Naming conventions help to describe processing and input/output. The maintenance programmer should be able to recognize global variables easily, since extra caution must be used when making changes which deal with data which is either generated or used outside the module being changed. Although the listings may not be available to confirm conventions, the documentation should contain standards or conventions to be followed during programming.

EXAMPLES: All variables which are global variables have names beginning with "XG" (external-global); no other type of variable name begins with that letter combination.

GLOSSARY: Global variable: Any variable or constant which can be accessed by more than one module of a program.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if there are no global variables.

QUESTION DATA SHEET

Question Number D-46

QUESTION: The terminology used in the documentation to describe the program is easily understood.

CHARACTERISTIC: Simplicity (format simplicity).

EXPLANATIONS: The general use of English and program terms should be simple, straightforward, easily understood; any terms or acronyms needing to be clarified should be defined prior to use and included in a glossary for reference.

EXAMPLES: A program that calculates MTBF should define Mean Time Between Failures - what the acronym means plus how the figure is calculated.

GLOSSARY: Terminology: Technical or special terms relevant to this particular computer system.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-47

QUESTION: The documentation is physically organized as a systematic description of the program from levels of less detail to levels of more detail.

CHARACTERISTIC: Simplicity (format simplicity).

EXPLANATIONS: Generally, the documentation produced during a software development effort should successively describe requirements, preliminary design, detailed design, operation/maintenance manual, test plan, etc. This will reflect a natural progression of program description from levels of less detail to levels of more detail.

EXAMPLES: Within any given documentation product, e.g., the detailed design specification, there should be a sequential progression from descriptions of less detail (e.g. overview) to descriptions of more detail (e.g., module design).

GLOSSARY: *Physically organized: The documents, volumes,* chapters, sections, etc.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-48

QUESTION: Each part (sentence, paragraph, subsection, section, chapter, volume, etc.) of the documentation tends to express one central idea.

CHARACTERISTIC: Simplicity (format simplicity).

EXPLANATIONS: All documentation should be scanned, If the documentation has been written in a simple understandable manner, then more than likely each part will address one primary topic (and subparts, one primary subtopic, etc.). The descriptions will be simple and to the point.

EXAMPLES:

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-49

QUESTION: The amount of cross referencing among parts of the documentation contributes to the understandability of the program description.

CHARACTERISTIC: Simplicity (format simplicity).

EXPLANATIONS: Some parts of the documentation may use cross referencing well while other parts may not. The evaluator should study the documentation until a reasonably well-founded overall impression is formed.

EXAMPLES: A narrative description of a file layout cross referenced to a figure graphically displaying the file is good. A simple reference to the figure with no narrative is not.

GLOSSARY: Cross reference: A note, statement, section number, etc. which directs a reader from one part of the documentation to another part.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-50

QUESTION: The documentation indicates that the program source language is a high order language (HOL).

CHARACTERISTIC: Simplicity (format simplicity).

EXPLANATIONS: Even though the system design dictates a non-HOL, a HOL is desirable from a maintainability standpoint. Less knowledge of internal machine operating characteristics is required to maintain a HOL program.

EXAMPLES: A particular communication processor program is better designed in assembly language due to the nature of bit manipulation requirements; however, assembly language programs are harder to maintain due to the machine dependency of assembly languages and the specialized knowledge required to maintain them.

GLOSSARY: High order language: A programming language that does not reflect the structure of any one given computer or that of any given class of computers: Non-assembly, non-micro code, non-machine; e.g., FORTRAN, JOVIAL, PL/I, PASCAL, etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if the program source language is completely HOL. Answer F if the program source language is completely non-HOL. Answer in the range B to E if the source language is a mix of HOL and non-HOL by approximate percentage:

B - $\geq$ 80%

C - $\geq$ 60%

D - $\geq$ 40%

E - $\geq$ 20%

QUESTION DATA SHEET

Question Number D-51

QUESTION: The documentation indicates that the use of recursive/reentrant programming techniques is not excessive.

CHARACTERISTIC: Simplicity (format simplicity).

EXPLANATIONS:

The documentation should identify within a general "program design considerations" section or the individual module description sections whether recursion or reentrancy is to be utilized. Many languages (or at least a particular implementation of a compiler) do not allow recursive and/or reentrant code. Some languages (e.g., stack-oriented languages like Pascal) allow recursion as a natural language capability.

The evaluator should get an overall impression of how much recursion/reentrant programming is a part of the overall program design. If done with care, some use of recursion or reentrancy can simplify the overall program design even though the particular modules which are recursive/reentrant will probably be harder to maintain because of those concerns.

EXAMPLES: Utility modules generally use these techniques.

GLOSSARY:

Recursive programming techniques: The use of operations which are defined in terms of themselves: a recursive module is one which uses a call to itself within the body of the code.

Reentrant programming technique: The technique of interrupting a program module at any point by another user and then resuming execution at the point of interruption. A reentrant module is one which can be concurrently used by more than one user.

Excessive: Detracts from simplicity.

SPECIAL RESPONSE INSTRUCTIONS:

If the documentation does not indicate, then:

Answer A if the language does not allow such techniques (example: COBOL).

Answer F if the language does allow such techniques (example: ALGOL or assembly language).

QUESTION DATA SHEET

Question Number D-52

QUESTION: The documentation indicates that each program module is designed to perform only one major function.

CHARACTERISTIC: Simplicity (design simplicity).

EXPLANATIONS: From the standpoint of simplicity, it would be easy to maintain a program in which each module performs only one function. Even if each module (or nearly each) performs only one major function and possibly one or two related functions, the program should still be simple and easy to maintain.

EXAMPLES: A print module may make a decision as to where to return in a program based upon the data printed. This may detract little from the simplicity; however, it would preclude an A answer.

GLOSSARY:

Function: A sub-division of a process.

SPECIAL RESPONSE INSTRUCTIONS: Answer A only if each module performs just one function. Answer B-F based on the proportion of modules which perform more than one function, e.g., B if 10% or less to F if 90% or more.

QUESTION DATA SHEET

Question Number D-53

QUESTION: The documentation indicates that resource (storage, timing, tape drives, disks, consoles, etc.) allocation is fixed throughout program execution.

CHARACTERISTIC: Simplicity (design simplicity).

EXPLANATIONS: Dynamic allocation tends to increase the level of complexity of a module, thereby making maintenance more difficult and time-consuming. The sharing or dynamic reassignment of resources should be a highlight of a section describing special processing (control) considerations, memory allocation, timing requirements by mission phase, etc. As another recourse, the evaluator can check the individual module descriptions for possible mention of any dynamic resource allocation.

EXAMPLES:

GLOSSARY:
Resource allocation: The assignment of a particular resource to a particular program task, function, module, etc.
Fixed: Is not reassigned from initialization to termination or reinitialization of the program.

SPECIAL RESPONSE INSTRUCTIONS: Answer A only if resource allocation is fixed throughout the entire program or is controlled by the operating system (i.e., transparent to the programmer).

QUESTION DATA SHEET

Question Number D-54

QUESTION: The documentation indicates that the control flow among modules is easy to follow.

CHARACTERISTIC: Simplicity (design simplicity).

EXPLANATIONS: The documentation should include narrative or a hierarchical flowchart which gives a clear, concise, easily understood general overview of the sequence in which modules (and perhaps submodules) are invoked and what controls that sequence.

EXAMPLES:

GLOSSARY: Control flow among modules: Which modules call and are called by other modules.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if all modules are entirely independent.

QUESTION DATA SHEET

Question Number D-55

QUESTION: The timing scheme designed for the program is easily understood from the documentation.

CHARACTERISTIC: Simplicity (design simplicity).

EXPLANATIONS: The program documentation should include a separate section which describes overall timing requirements and the timing scheme designed to satisfy those requirements. This description should be clear, concise, and easily followed.

EXAMPLES:

GLOSSARY: Timing scheme: Time slicing, time sharing, priority levels, rate groups, etc. as applied to the overall sequencing and execution of program functions.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if there are clearly no special timing considerations.

QUESTION DATA SHEET

Question Number D-56

QUESTION: The program is designed so that modules are not interrupted during execution.

CHARACTERISTIC: Simplicity (design simplicity).

EXPLANATIONS: Whenever special processing is required to handle the possibility of being interrupted, a higher level of complexity will exist in a module.

EXAMPLES:

GLOSSARY: Interrupted: Execution is suspended without the knowledge of the module being suspended.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-57

QUESTION: It is evident from the documentation that a knowledge of mathematics beyond basic algebra is not required to understand the mathematical functions performed by the program.

CHARACTERISTIC: Simplicity (design simplicity).

EXPLANATIONS: There may be a few complex functions, but on the average most of the functions require no mathematics beyond basic algebra. In this case, the evaluator might generally or strongly agree with the question statement. If there appears to be many complex functions, the evaluator may want to generally or strongly disagree with the question statement.

EXAMPLES:

GLOSSARY: Basic algebra: Functions (including trigonometric and geometric functions), equations, polynomials (including series), graphing of functions, basic manipulations, etc.; excludes calculus, differential equations, Fourier transforms, statistical algorithms, etc.

SPECIAL RESPONSE INSTRUCTIONS: The evaluator should respond on the basis of overall program considerations.

QUESTION DATA SHEET

Question Number D-58

QUESTION: A numbering scheme has been adopted which allows for easy addition or deletion of narrative parts of the documentation.

CHARACTERISTIC: Expandability (format expandability).

EXPLANATIONS: Computer program documentation can be voluminous and subject to frequent changes due to program modifications and format requirement alterations. This question seeks to determine if:

a) A numbering convention has been established for formating the documentation; and,

b) the format enhances:

1 identifying volumes, sections, and paragraphs; and pages.

2 adding and deleting information without generating attendant rippling effects throughout the rest of the document. Determine if a numbering scheme has been established. Assess the ease with which a volume/section/paragraph/page can be located and the extent to which a change in document content will affect the numerical identifiers of other parts of the document.

EXAMPLES: Consecutive numbering of pages makes it difficult to and/delete pages. Use of a hierarchical numbering system to number pages by section reduces the number of succeeding pages affected by changing the contents of a section.

GLOSSARY: Number scheme: A formatting convention used to facilitate identifying some part of a document.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-59

QUESTION: Graphic materials (figures, charts, lists, etc.) are physically separate (e.g., on separate pages) from narrative description.

CHARACTERISTIC: Expandability (format expandability).

EXPLANATIONS: Graphic materials should always be on separate pages. Changes in narrative are more easily typed when narrative and graphic materials are not co-located on the same page.

EXAMPLES:

GLOSSARY: Graphic materials: Tables, figures, equations.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-60

QUESTION: A numbering scheme has been adopted which allows for easy addition or deletion of graphic materials.

CHARACTERISTIC: Expandability (format expandability).

EXPLANATIONS: Graphic materials in computer program documentation can be subject to frequent changes due to program or requirement modifications. A suitable numbering scheme should have been established such that graphic materials can easily be identified and added/removed without having a rippling effect on other numbered items in the document. It should be determined if a numbering scheme has been established. The ease with which graphic materials can be located and the extent to which adding or deleting an item affects the assigned identifiers of other items should be assessed.

EXAMPLES: Consecutive numbering of figures across major sections requires more changes when adding or deleting figures than numbering consecutively within a major section.

GLOSSARY:

Numbering scheme: A formatting convention used to facilitate identifying some part of a document.

Graphic materials: Items such as tables, figures, and equations.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-61

QUESTION: The program timing scheme appears to be flexible enough to allow for modifications (e.g., reorganization, addition, deletion of functional parts).

CHARACTERISTIC: Expandability (design expandability).

EXPLANATIONS: In many applications, specific program functions must be performed at periodic intervals, within predetermined time intervals, or at a definite point in time. The question seeks to determine the extent to which the program's timing scheme restricts desired changes to a program's design.

EXAMPLES: A function that must be performed every 10 microseconds will conflict with the design of a different function requiring 10 or more microseconds of uninterrupted processing.

GLOSSARY:

Timing scheme: A convention based on wall clock time or processor clock time that controls execution of a program's functions.

Flexible: Modifiable.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if there are no required program timing considerations.

QUESTION DATA SHEET

Question Number D-62

QUESTION: There is a reasonable time margin for each major program function (rate group, time slice, priority level, etc.).

CHARACTERISTIC: Expandability (design expandability).

EXPLANATIONS: Program functions should be designed such that required timing constraints are met with "room to spare." Too little reserves limit the ability to add processes to a function. Too much reserve, on the other hand, may indicate processing inefficiency due to resource underutilization.

EXAMPLES: A program function requiring a periodic 5 milli-second time slice is allocated a dedicated 20 millisecond time slice. The timing margin for this function is 75%.

GLOSSARY:

Timing margin: A percentage of the time allocated to a process that is still available for use; calculated by the ratio of spare time to the total time frame.

Program function: A generic term used to reference one or more program processes.

Time slice: A predetermined period of processer time.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if the program has no timing requirements because timing margins will not be of any concern (e.g., program in non-real time). Also answer A if each timing margin is at least 25%.

QUESTION DATA SHEET

Question Number D-63

QUESTION: Documentation narrative explains the procedures for altering basic data storage sizes.

CHARACTERISTIC: Expandability (design expandability).

EXPLANATIONS: How to alter the capacity of data storage is not always obvious. Very often, storage has been judiciously allocated to interface with various portions of the program. Documentation narrative should not only describe how to alter basic data storage sizes, but should also identify those interfaces which might be impacted by such changes.

EXAMPLES: Creating a new variable in the middle of a labeled common region can affect all program processes that use that storage area.

GLOSSARY: Basic data storage sizes: The size of program data structures upon which program processing depends; the structure may be an array, a table, space allocated by an assembly directive, etc.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-64

QUESTION: The program has been designed to allow for an increase in storage utilized before storage capacity is exceeded.

CHARACTERISTIC: Expandability (design expandability).

EXPLANATIONS: Over time, the amount of data storage space required for program applications almost always increases. A program should be designed so that additional storage allocations can be made without the need for program design or hardware modification.

EXAMPLES:

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Answer A if at least 25% of the storage capacity is available for future use.

QUESTION DATA SHEET

Question Number D-65

QUESTION: Those modules dependent upon data structure sizes are identified.

CHARACTERISTIC: Expandability (design expandability).

EXPLANATIONS: Changing the definition of a data structure will invariably impact the modules that use it. Therefore, the documentation should contain a list of "affected modules" for each data structure so that changes to the structure can be accompanied by appropriate changes to the modules.

EXAMPLES:

GLOSSARY:
Data Structure: Grouping of data elements (variables and constants) into arrays, records, files, etc.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if it is clear that no modules are dependent upon data structure definitions. This will be unlikely in large software programs.

QUESTION DATA SHEET

Question Number D-66

QUESTION: The program has been designed so that functional parts may be easily added or deleted.

CHARACTER"STIC: Expandability (design expandability).

EXPLANATIONS: Programs designed using a top-down, structured methodology often consist of functional parts which are interrelated, yet independent, of one another. That is, each part can be viewed as a "black box" externally. Such parts are usually easily added, deleted, or replaced. However, functional parts designed with complicated, delicate interfaces are more difficult to deal with. An impression should be formed from the module descriptions and program overview information whether the functional parts could be easily added or deleted.

EXAMPLES: Functions executed as a result of a table-driven executive can be easily added and removed by modifying the contents of the table.

GLOSSARY: Functioal parts: Primarily modules, but also includes groups of modules that perform major functions.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-67

QUESTION: There is a separate part of the documentation for the description of a program test plan.

CHARACTERISTIC: Instrumentation (format instrumentation).

EXPLANATIONS: Testing is generally regarded as a separate organizational function. It is helpful to those individuals invloved in testing to have test information gathered into one part of the documentation.

EXAMPLES: The documentation may include volumes of test information sheets. It may include test plans; acceptance test procedures (ATP), formal or preliminary qualification test (FQT, PQT) procedures. It may include sets of sample input data with expected output data.

GLOSSARY: Program test plan: Set of descriptions and procedures for how the program is to be (or can be, or has been) tested.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if a separate part exists. Answer F if the description does not exist. If for some reason the program test plan description is distributed over several separate parts (e.g., one part per unit/module description), then answer in the range B to E as to the effectiveness of that "separation" from the point of view of program test/retest.

QUESTION DATA SHEET

Question Number D-68

QUESTION: There is a separate part of the documentation for the description of sample test data.

CHARACTERISTIC: Instrumentation (format instrumentation).

EXPLANATIONS: Comparison of input/output data before and after program changes have been made is one of the best ways to assure that changes have been made properly and that no extraneous errors have been introduced.

EXAMPLES:

GLOSSARY: Sample test data: The input and output data used for the program tests.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if a separate part exists. Answer F if the description does not exist. If for some reason the sample test data description is distributed over several separate parts (e.g., one part per unit/module description), then answer in the range B to E as to the effectiveness of that "separation" from the point of view of program test/retest.

QUESTION DATA SHEET

Question Number D-69

QUESTION: There is a separate part of the documentation for the description of program support tools which would aid in testing the program.

CHARACTERISTIC: Instrumentation (format instrumentation).

EXPLANATIONS: Program support tools are not generally a part of the operational software. Descriptions of program support tools are often voluminous and would merely lead to confusion if they were included with descriptions of the operational software. However, the descriptions of program support tools are absolutely necessary and should therefore constitute a separate part of the documentation.

EXAMPLES: A FORTRAN reference manual is an absolute necessity to a scientific programmer, but is definitely not considered to be an integral part of applications software documentation.

GLOSSARY: Program support tools: General debug aids, test/retest software, trace software/hardware features, use of compiler/link editor/library management/configuration management/text editor/display software tools.

SPECIAL RESPONSE INSTRUCTIONS: Answer A if a separate part exists. Answer F if the description does not exist. If for some reason the description of program support tools for testing is distributed over several separate parts then answer in the range B to E as to the effectiveness of that "separation" from the point of view of program test/retest.

QUESTION DATA SHEET

Question Number D-70

QUESTION: A set of test procedures to be used for program check-out are explained.

CHARACTERISTIC: Instrumentation (design instrumentation).

EXPLANATIONS: Program test procedures, in order to be useful, must provide adequate information to completely describe test inputs, outputs, and environment.

EXAMPLES: One good test of the adequacy of the explanation of the program test procedures is for the evaluator to visualize how easy it would be to execute step-by-step one or more of the particular test procedures. If the information is not presented in a step-by-step fashion with a complete discussion of the test environment, test inputs and expected test outputs, then the test will probably be difficult to perform.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: If no test procedures exist, then answer "F."

QUESTION DATA SHEET

Question Number D-71

QUESTION: The set of test procedures provides useful unit testing information.

CHARACTERISTIC: Instrumentation (design instrumentation).

EXPLANATIONS: The test procedures will ordinarily be described in terms of unit testing information and integration testing information. The test procedures should describe test procedures for sub-units of the program as well as for overall program testing. It is often infeasible to test the entire program during modification/testing of only one sub-unit.

EXAMPLES:

GLOSSARY: Unit: Units may be modules, submodules, groups of modules or some other organization depending upon the contractor and the application area.

SPECIAL RESPONSE INSTRUCTIONS: If no test procedures exist, then answer "F."

QUESTION DATA SHEET

Question Number D-72

QUESTION: The set of test procedures provides useful information on limitations/incompleteness.

CHARACTERISTIC: Instrumentation (design instrumentation).

EXPLANATIONS: The testing agency, in order to know what was actually tested and to what extent it was tested, must know the limitations of test procedures.

EXAMPLES: The documentation should contain the ranges of variables tested and not tested as well as modules tested and not tested.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: If no test procedures exist or there is no information on the limitations/incompleteness of the test procedures, then answer F.

QUESTION DATA SHEET

Question Number D-73

QUESTION: The program has been designed with the capability to display test inputs and outputs in summary form.

CHARACTERISTIC: Instrumentation(design instrumentation).

EXPLANATIONS: Many programs process tremendous quantities of data and the test inputs/outputs likewise consist of tremendous quantities of data. In such cases, it is desirable to have a program automatically compare the test data and display only the differences/errors to the maintainer.

EXAMPLES:

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: If the module does not process a great deal of data, so that there is no need to summarize the test inputs/outputs, your answer should lie in the B-E range.

QUESTION DATA SHEET

Question Number D-74

QUESTION: The documentation describes a standardized set to program test data (input and output) that has been designed to exercise the program.

CHARACTERISTIC: Instrumentation (design instrumentation).

EXPLANATIONS: This question relates to both quality and existence of test data. In order to assure that test data properly exercises or tests the program, it must be carefully designed to do so. Randomly assembled data will not usually exercise all parts of the program, whereas carefully designed test data will.

EXAMPLES:

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-75

QUESTION: The documentation indicates that the program has been designed to include software test probes to aid in identifying processing performance.

CHARACTERISTIC: Instrumentation (design instrumentation).

EXPLANATIONS: Test data alone is usually not sufficient to adequately test a program. Certain parts of the program can only be tested by insertion (or activation) of special executable code which is used strictly for testing purposes.

EXAMPLES: If the language provides debug capabilities or such options as conditional compilation, then the designer is much more likely to consider the use of test probes as a normal part of the program. However, it is still possible under more adverse conditions for the design to include separate functions which can be individually invoked for the purpose of collecting appropriate processing performance information.

GLOSSARY:

Include: Presently in-line or can be inserted in-line through activation.

Software test probe: Section of code or special module which collects certain process parameters; generally the activation of the probe can be controlled through user options.

Processing performance: Accuracy, timing, etc.

SPECIAL RESPONSE INSTRUCTIONS:

QUESTION DATA SHEET

Question Number D-76

QUESTION: Error checking within the program has been designed to include such features as diagnostic reporting, I/O parameter checking, runtime index range checking, etc.

CHARACTERISTIC: Instrumentation (design instrumentation).

EXPLANATIONS: These particular test tools, as well as many others, are of particular importance to program instrumentation and test.

EXAMPLES: The documentation describing error processing/error codes/error messages, or perhaps report generation could be checked to determine what type of error checking appears to be done. In addition, the general design conventions/ standards might indicate what error checking conventions have ben adopted. The source language compiler may have options which allow for the generation of run-time parameter checking (e.g., index range).

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: The evaluator must judge which particular types of instrumentation he feels should be included, and answer A - F according to his estimation of the adequacy of what actually exists.

QUESTION DATA SHEET

Question Number D-77

QUESTION: Modularity as reflected in the program documentation contributes to the maintainability of the program.

CHARACTERISTIC: General questions.

EXPLANATIONS: Software possesses the characteristics of modularity to the extent a logical partitioning of software into parts, components, modules has occurred.

EXAMPLES: The software has been partitioned into easily comprehendable "sections." Each "section" is independent from every other "section" as much as is reasonable; i.e., to understand any given "section," requisite knowledge of other "sections" has been kept to a minimum.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Please give your general feeling about the modularity of the documentation.

QUESTION DATA SHEET

Question Number D-78

QUESTION: Descriptiveness as reflected in the program documentation contributes to the maintainability of the program.

CHARACTERISTIC: General questions.

EXPLANATIONS: Software possesses the characteristics of descriptiveness to the extent that it contains information regarding its objectives, assumptions, inputs, processing, outouts, components, revision status, etc.

EXAMPLES: Program objectives are explained, subprogram objectives are explained, communication links are either specifically explained or there is a detailed plan for setting up the communication links. Revision status of the documentation is clear. Source listing revision status is either clear or a detailed plan for revision status tracking is explained, etc.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Please give your general feeling about the descriptiveness of the documentation.

QUESTION DATA SHEET

Question Number D-79

QUESTION: Consistency as reflected in the program documentation contributes to the maintainability of the program.

CHARACTERISTIC: General questions.

EXPLANATIONS: Software possesses the characteristics of consistency to the extent the software products correlate and contain uniform notation, terminology and symbology.

EXAMPLES: Things are done similarly in different parts of the documentation. Once an individual learns how the documentation is set up, he can turn to any part of the documentation and see exactly what he expects to see. A set of documentation standards appears to have been set up and followed.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Please give your general feelings about the consistency of the documentation.

QUESTION DATA SHEET

Question Number D-80

QUESTION: Simplicity as reflected in the program documentation contributes to the maintainability of the program.

CHARACTERISTIC: General question.

EXPLANATIONS: Software possesses the characteristics of simplicity to the extent that it lacks complexity in organization, language, and implementation techniques and reflects the use of singularity concepts and fundamental structures.

EXAMPLES: The organization of the documentation is logical. Uncomplicated, descriptive terminology is used throughout. Each section or part of the documentation addresses a single subject and is minimally dependent upon other parts for a full understanding.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Please give your general impression about the simplicity of the documentation.

QUESTION DATA SHEET

Question Number D-81

QUESTION: Expandability as reflected in the program documentation contributes to the maintainability of the program.

CHARACTERISTIC: General question.

EXPLANATIONS: Software possesses the characteristics of expandability to the extent that a physical change to information, computational functions, data storage or execution time can be easily accomplished.

EXAMPLES: The documentation contains standards for programming which enhance expandability of the code.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Please give your general impression of the overall expandability of the documentation and the program design as reflected in the documentation.

QUESTION DATA SHEET

Question Number D-82

QUESTION: Instrumentation as reflected in the program documentation contributes to the maintainability of the program.

CHARACTERISTIC: General questions.

EXPLANATIONS: Software possesses the characteristics of instrumentation to the extent it contains aids which enhance testing.

EXAMPLES: The documentation contains test cases which slow known input and expected output. The documentation also contains a plan or standards for program instrumentation. Some sort of DEBUG mode execution is specifically addressed.

GLOSSARY:

Debug:	Removal of bugs.
Bug(s):	Latent error(s).

SPECIAL RESPONSE INSTRUCTIONS: Please give your feelings about the instrumentation of the software as reflected in the documentation.

QUESTION DATA SHEET

Question Number D-83

QUESTION: Overall it appears that the characteristics of the program documentation contribute to the maintainability of the program.

CHARACTERISTIC: General questions.

EXPLANATIONS: Software maintainability is a quality of software which is defined as those characteristics which affect the ability of the software engineers to:

1) Correct errors.
2) Add system capabilities through software changes.
3) Delete features.
4) Modify software to be compatible with hardware changes.

EXAMPLES: The program documentation is designed to aid you in maintenance of the subject software. It is not after-the-fact documentation except in those cases where it should be.

GLOSSARY:

SPECIAL RESPONSE INSTRUCTIONS: Please give your general impression as to how well the documentation would aid you in maintenance of the software under study.

LIST OF REFERENCES

1. Boehm, B. W., "Software and Its Impact: A Quantitative Assessment", Datamation, Vol 19, No. 5, May, 1973.

2. U. S. General Accounting Office, Report to the Congress, Contracting For Computer Software Development - Serious Problems Require Management Attention to Avoid Wasting Additional Millions, Report Number FGSMD-80-4, November 9, 1979.

3. U. S. Department of Commerce, NBS Special Publication 500-7, "Computers in the Federal Government: A Compilation of Statistics", June, 1977, pp. viii, 24-28.

4. Boehm, B. W., "Software Engineering", IEEE Transactions on Computers, Vol C-25, No. 12, IEEE, December, 1976.

5. U. S. General Accounting Office, Report to the Congress, Federal Agencies Maintenance of Computer Programs: Expensive and Undermanaged, Report Number AFMD-81-25, February 26, 1981.

6. Coppola, A. W. and Sukert, A. N., Reliability and Maintenance Management Manual, Rome Air Development Center Report RADC-TR-79-200, July, 1979.

7. De Roze, B. C., Special Presentation, Proceedings of the "Managing the Development of Weapons System Software Conference", May, 1976.

8. Mills, H. D., "Software Development", IEEE Transactions on Software Engineering, Vol SE-2, No. 4, IEEE, December, 1976.

9. Elshoff, J. L., "An Analysis of Some Commercial PL/1 Programs", IEEE Transactions on Software Engineering, Vol SE-2, No. 2, IEEE, June, 1976.

10. Daly, E. B., "Management of Software Development", IEEE Transactions on Software Engineering, Vol. SE-3, No. 2, IEEE, May, 1977.

11. Naur, Peter and Randall, Brian, Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmish, Germany, 7-11 October 1968.

12. Naval Research Laboratory, Software Engineering Principles, Notes from a Course Offered at the Naval Postgraduate School, Monterey, Ca., 3-14 August 1981.

13. Gilb, Tom, Software Metrics, Winthrop Publishing, Inc., 1977.

14. Gill, S., "The Origins and Meaning of Software Engineering", Software Engineering, Software Engineering International State of the Art Report, Infotech, 1972.

15. Manley, J. H., "Embedded Computer Systems", Findings and Recommendations of the Joint Logistics Commanders Software Reliability Work Group, Vol. 2, 1975.

16. Bauer, F. L., "Software Engineering", Information Processing 71, North-Holland, 1972.

17. Jensen, R. W. and Tonies, C. C., Software Engineering, Prentice-Hall, 1979.

18. Hoare, C. A. R., "Software Engineering: A Keynote Address", Proceedings of the 3rd International Conference on Software Engineering, IEEE, 10-12 May 1978.

19. Wasserman, A. I. and Freeman, P., "Software Engineering Education: Status and Prospects", Proceedings of the IEEE, Vol. 66, No. 8, IEEE, August, 1978.

20. Wasserman, A. I. and others, "Essential Elements of Software Engineering Education", Proceedings of the 2nd International Conference on Software Engineering, IEEE, 13-15 October 1976.

21. Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, 1975.

22. Wulf, W. A., "Programming Methodology", Proceedings of a Symposium on the High Cost of Software, J. Goldberg (ed.), Stanford Research Insitutue, September, 1973.

23. Ross, D. T. and others, "Software Engineering: Process Principles, and Goals", Tutorial on Software Design Techniques, Freeman, P. and Wasserman, A. I. (eds.), IEEE, 1980.

24. Boehm, B. W. and others, Characteristics of Software Quality, North-Holland, 1978.

25. Littlewood, B., "How to Measure Software Reliability and how not to", Proceedings of the 3rd International Conference on Software Engineering, IEEE, 10-12 May 1978.

26. Kline, M. B. and Schneidewind, N. F., "Life Cycle Comparisons of Hardware and Software Maintainability", Proceedings of the Third National Reliability Conference-Reliability 81, 1981.

27. Kline, M. B., "Hardware and Software Reliability and Maintainability: What are the Differences", Proceedings, 1980 Annual Reliability and Maintainability Symposium, IEEE, January, 1980.

28. Freeman, P., "Software Reliability and Design: A Survey", Proceedings, 13th Design Automation Conference, IEEE, June, 1976.

29. Dijkstra, E. W., "Notes on Structured Programming", Structured Programming, Academic Press, 1972.

30. Parnas, D. L., "The Use of Precise Specifications in the Development of Software", Proceedings of the IFIP 1977, 1977.

31. Liskov, B. H., "A Design Methodology for Reliable Software", Proceedings, Fall Joint Computer Conference, AFIPS Press, 1972.

32. Glass, R. L., Software Reliability Guidebook, Prentice-Hall, 1979.

33. Parnas, D. L. and Wurges, H., "Responses to Undesired Events in Software Systems", Proceedings, 2nd International Conference on Software Engineering, IEEE, 13-15 October 1976.

34. Wasserman, A. I., "Information System Design Methodology", Journal of the American Society for Information Science, Vol. 31, No. 1, January, 1980.

35. Bradley, G. H. and others, Structure and Error Detection in Computer Software, NPS55ss75021, Naval Postgraduate School, February, 1975.

36. Schneidewind, N. F. and others, System Test Methodology, Vols I, III, NPS55ss75072, Naval Postgratuate School, July, 1975.

37. McCabe, T., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2, No. 4, IEEE, December, 1976.

38. Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", Communications of the ACM, Vol. 9, No. 5, Association for Computing Machinery, May, 1966.

39. Lientz, B. P. and Swanson, E. B., Software Maintenance Management, Addison-Wesley, 1980.

40. Swanson, E. B., "The Dimensions of Maintenance", Proceedings of the 2nd International Conference on Software Engineering, IEEE, 13-15 October 1976.

41. Myers, G. J., Software Reliability Principles and Practices, John Wiley & Sons, 1976.

42. Tausworthe, R. C., Standardized Development of Computer Software, (Part I, Methods, Part II, Standards), Jet Propulsion Laboratory, California Institute of Technology, Part I, 1976, Part II, 1978.

43. Poole, P. C. and Waite, W. M., "Portability and Adaptability", Advanced Course on Software Engineering, Springer-Verlag, 1973.

44. Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resources Management, Final Report: Joint Logistics Commanders Software Workshop Panel E: Software Reusability, Monterey, Ca., 22 June 1981.

45. Glass, R. L., "From Pascal to Pebbleman and Beyond", Datamation, Vol. 25, No. 7, July, 1979.

46. Fisher, D. A., "DOD's Common Programming Language Effort", Computer, Vol. 11, No. 3, March, 1978.

47. Kijkstra, E. W., "On the Green Language Submitted to the DoD", SIGPLAN Notices, October, 1978.

48. Parnas, D. L., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, IEEE, March, 1979.

49. Parnas, D. L., "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, IEEE, March, 1976.

50. Thalmann, D., "Evolution in the Design of Abstract Machines for Software Portability", Proceedings of the 3rd International Conference on Software Engineering, IEEE, 10-12 May 1978.

51. Buxton, J. N. and Randell, B., (eds.), Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 October 1969.

52. Brooks, F. P., The Mythical Man-Month, Addison-Wesley, 1975.

53. Wulf, W. A., "The Case Against the GOTO", Proceedings of the 25th ACM National Conference, Vol. 2, 1972.

54. Myers, G. J., "Characteristics of Composite Design", Datamation, Vol. 19, No. 9, September, 1973.

55. Myers, G. J., "Composite Design: The Design of Modular Programs", Technical Report TR00.2406, IBM, January 29, 1973.

56. Myers, G. J., Reliable Software Through Composite Design, Petrocelli/Charter, 1975.

57. Yourdon, E. and Constantine, L. L., Structured Design, Yourdon Press, 1978.

58. Parnas, D. L., "Information Distribution Aspects of Design Methodology", Proceedings of the IFIP Congress 71, North-Holland, 1971.

59. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, Association for Computing Machinery, December, 1972.

60. Heninger, K. L., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", Proceedings, Conference on Specification of Reliable Software, IEEE, 1979.

61. Heninger, K. L., Parnas, D. L., and others, Software Requirements for the A-7 Aircraft, Naval Research Laboratory Memorandum Report 3876, November 27, 1978.

62. Parnas, D. L., Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems, Naval Research Laboratory Report 8047, 1977.

63. Baker, F. T., "Chief Programmer Team Management of Production Programming", IBM Systems Journal, Vol. 11, No. 2, January, 1972.

64. Wirth, N., "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No. 4, Association for Computing Machinery, April, 1971.

65. McClure, C. L., "Top-Down, Bottom-Up and Structured Programming", Proceedings of the 1st International Conference on Software Engineering, IEEE, 11-12 September 1975.

66. Dijkstra, E. W., "The Structure of 'THE'-Multiprogramming System", Communications of the ACM, Vol. 11, No. 5, Association for Computing Machinery, May, 1968.

67. Schneidewind, N. F., Software Maintenance: Improvement Through Better Development Standards and Documentation, NPS-54-82-002, Naval Postgraduate School, Monterey, Ca., 22 February 1982.

68. Ramamoorthy, C. V. and Ho, S. B. F., "Testing Large Software with Automated Software Evaluation Systems', IEEE Transactions on Software Engineering, SE-1, No. 1, IEEE, March, 1975.

69. Branstad, M. A. and others, NBS Special Publication 500-56, "Validation, Verification and Testing for the Individual Programmer", U. S. Department of Commerce, February, 1980.

70. Weinberg, G. M., The Psychology of Computer Programming, Van Nostrand Reinhold, 1971.

71. Stanfield, J. R. and Skrukrud, A. M., Software Acquisition Management Guidebook, Software Maintenance Volume, System Development Corporation, TM-5772-004-02, October, 1977.

72. Yourdon, E. N., (ed.), Classics in Software Engineering, Yourdon Press, 1979.

73. Gordon, R. L. and Lamb, J. C., "A Close Look at Brook's Law", Datamation, Vol. 23, No. 6, June, 1977.

74. Teichroew, D., and Hershey, E. A., "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information", IEEE Transactions on Software Engineering, SE-3, No. 1, IEEE, January, 1977.

75. Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Systems", IEEE Transactions on Software Engineering, SE-3, No. 1, IEEE, January, 1977.

76. Bell, T. M. and others, "An Extended Approach to Computer-Aided Software Engineering Requirements", IEEE Transactions on Software Engineering, SE-3, No. 1, IEEE, January, 1977.

77. Petrie, F. A., The Utilization of Requirement Statement Methodologies in the United States Navy and Their Impact on Systems Acquisition, Master's Thesis, Naval Postgraduate School, Monterey, Ca., March, 1980.

78. Secretary of the Navy Instruction 3560.1, Navy Tactical Digital Systems Documentation Standards, 8 August 1974.

79. Department of Defense Military Standard 490, Specification Practices, 30 October 1968.

80. U. S. Department of Commerce, (National Bureau of Standards), Federal Information Processing Standard 38, "Guidelines for Documentation of Computer Programs and Automated Data Systems", 15 February 1976.

81. DeMarco, T., Structured Analysis and Systems Specification, Yourdon Press, 1979.

82. DeMarco, T., "Structured Analysis and Systems Specification", Classics in Software Engineering, Yourdon, E. N. (ed.), Yourdon Press, 1978.

83. Caine, S. H. and Gordon, E. K., "PDL - a Tool for Software Design", Proceedings of the 1975 National Computer Conference, Vol. 44, AFIPS, 1975.

84. Kirk, H. W., "Use of Decision Tables in Computer Programming", Communications of the ACM, Vol. 8, No. 1, Association for Computing Machinery, January, 1965.

85. Pooch, U. W., "Translation of Decision Tables", Computing Surveys, Association for Computing Machinery, June, 1974.

86. Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions on Software Engineering, SE-3, No. 1, IEEE, January, 1977.

87. Ross, D. T., and Shoman, K. E., "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, SE-3, No. 1, IEEE, January, 1977.

88. Jones, C., "A Survey of Programming Design and Specification Techniques", Proceedings, Conference on Specifications of Reliable Software, IEEE, 1974.

89. Stevens, W. P. and others, "Structured Design", IBM Systems Journal, Vol. 13, No. 2, IBM, 1974.

90. Fitzgerald, J. F., and others, Fundamentals of Systems Analysis, John Wiley & Sons, 1981.

91. Peters, L. J., and Tripp, L. L., "Comparing Software Design Methodologies", Datamation, Vol. 23, No. 11, November, 1977.

92. Jackson, M. A., Principles of Program Design, Academic Press, 1975.

93. Jackson, M. A., "Constructive Methods of Design", Tutorial on Software Design Techniques, Wasserman, A. I. and Freeman, P. (eds.), IEEE, 1980.

94. Riddle, W. E., "An Event-Based Design Methodology Supported by Dream", Formal Models, and Practical Tools for Information Systems Design, Schneider, H. J., (ed.), North-Holland, 1979.

95. Hamilton, M., and Zeldin, S., "Higher Order Software - A Methodology for Designing Software", IEEE Transactions on Software Engineering, SE-2, No. 1, IEEE, March, 1976.

96. International Business Machines Corporation, HIPO - A Design Aid and Documentation Technique, GC20-1851-1, 1974.

97. Stay, J. F., "HIPO and Integrated Program Design", IBM Systems Journal, Vol. 15, No. 2, IBM, 1976.

98. U. S. Department of Commerce, (National Bureau of Standards), Federal Information Processing Standard 24, "Flowcharting Symbols and Their Usage in Information Processing", 30 June 1973.

99. Nassi, I., and Schneiderman, B., "Flowchart Techniques for Structured Programming", SIGPLAN Notices, Vol. 8, Association for Computing Machinery, August, 1973.

100. Yoder, C. M., and Schrag, M. L., "Nassi-Shneiderman Charts - An Alternative for Design", Proceedings of the Software Quality and Assurance Workshop, Association for Computing Machinery, November, 1978.

101. Anderson, G. E., and Shumate, K. C., "Documentation Study Proves Utility of Program Listings", Computerworld, May 21, 1979.

102. "Chief Programmer Teams: Principles and Procedures", Report No. FSC 71-5108, IBM, Federal Systems Division, Gaithersburg, Md., June, 1971.

103. Bersoff, E. H., and others, "Software Configuration Management: A Tutorial", Computer, Vol. 12, No. 1, IEEE, January, 1979.

104. Department of Defense Directive 5000.29, Management of Computer Resources in Major Defense Systems, 26 April 1976.

105. Department of Defense Military Standard 1679, Weapons System Software Development, 1 December 1978.

106. Department of Defense Military Standard 52779 (AD), Software Quality Assurance Program Requirements, 5 April 1974.

107. Lindhorst, W. M., "Scheduled Maintenance of Applications Software", Datamation, Vol. 19, No. 5, May, 1973.

108. Pilcher, R. D., "Techniques Available for Improving the Maintainability of DOD Weapon System Software", Master's Thesis, Naval Postgraduate School, June 1980.

109. Young, R. A., "Life Cycle Concepts and Documentation Types", Documentation of Computer Programs and Automated Data Systems, National Bureau of Standards Special Publication 500-15, July, 1977.

110. Peters, L., "A Conceptional Basis for Software Design Standards", IEEE Software Engineering Standards Application Workshop, IEEE No. 81ch1633-7, August, 1981.

111. Federal Information Processing Standards Publication 38, Guidelines for Documentation of Computer Programs and Automated Data Systems, February, 1976.

112. Software OT&E Guidelines, Software Maintainability Evaluator's Handbook, Vol. III, Air Force Test and Evaluation Center, Kirtland AFB, New Mexico, April, 1980.

113. Software OT&E Guidelines, Guide for the Deputy for Software Evaluation, Vol. II, Air Force Test and Evaluation Center, Kirtland AFB, New Mexico, December, 1981.